Chapter 6

Functions upon Functions upon Functions

Often we need to apply a function to every element of a list. For example, doubling each of the numbers in a list of integers. We could do this with a simple recursive function, working over each element of a list:

```
double : int list → int list

let rec double l =
  match l with
    [] -> []                                    no element to process
  | h::t -> (h * 2) :: double t       process the element, and the rest
```

For example,

$$\begin{aligned}
& \underline{\texttt{double [1; 2; 4]}} \\
\Longrightarrow\quad & \texttt{2 :: }\underline{\texttt{double [2; 4]}} \\
\Longrightarrow\quad & \texttt{2 :: 4 :: }\underline{\texttt{double [4]}} \\
\Longrightarrow\quad & \texttt{2 :: 4 :: 8 :: }\underline{\texttt{double []}} \\
\Longrightarrow\quad & \underline{\texttt{2 :: 4 :: 8 :: []}} \\
\overset{*}{\Longrightarrow}\quad & \texttt{[2; 4; 8]}
\end{aligned}$$

The result list does not need to have the same type as the argument list. We can write a function which, given a list of integers, returns the list containing a boolean for each: `true` if the number is even, `false` if it is odd.

```
evens : int list → bool list

let rec evens l =
  match l with
    [] -> []                                    no element to process
  | h::t -> (h mod 2 = 0) :: evens t  process the element, and the rest
```

For example,

$$\begin{array}{ll}
 & \underline{\texttt{evens [1; 2; 4]}} \\
\Longrightarrow & \texttt{false :: }\underline{\texttt{evens [2; 4]}} \\
\Longrightarrow & \texttt{false :: true :: }\underline{\texttt{evens [4]}} \\
\Longrightarrow & \texttt{false :: true :: true :: }\underline{\texttt{evens []}} \\
\Longrightarrow & \underline{\texttt{false :: true :: true :: []}} \\
\overset{*}{\Longrightarrow} & \texttt{[false; true; true]}
\end{array}$$

It would be tedious to write a similar function each time we wanted to apply a different operation to every element of a list – can we build one which works for any operation? We will add a function as an argument too:

```
map : (α → β) → α list → β list

let rec map f l =
  match l with
    [] -> []                                        no element to process
  | h::t -> f h :: map f t               process the element, and the rest
```

The `map` function takes two arguments: a function which processes a single element, and a list. It returns a new list. We will discuss the type in a moment. For example, if we have a function `halve`:

```
halve : int → int

let halve x = x / 2
```

We can use map like this:

$$\begin{array}{ll}
 & \underline{\texttt{map halve [10; 20; 30]}} \\
\Longrightarrow & \texttt{5 :: }\underline{\texttt{map halve [20; 30]}} \\
\Longrightarrow & \texttt{5 :: 10 :: }\underline{\texttt{map halve [30]}} \\
\Longrightarrow & \texttt{5 :: 10 :: 15 :: }\underline{\texttt{map halve []}} \\
\Longrightarrow & \underline{\texttt{5 :: 10 :: 15 :: []}} \\
\overset{*}{\Longrightarrow} & \texttt{[5; 10; 15]}
\end{array}$$

Now, let us look at that type: $(\alpha \to \beta) \to \alpha$ **list** $\to \beta$ **list**. We can annotate the individual parts:

$$\overbrace{(\alpha \to \beta)}^{\text{function f}} \to \overbrace{\alpha\ \textbf{list}}^{\text{argument list}} \to \overbrace{\beta\ \textbf{list}}^{\text{result list}}$$

We have to put the function `f` in parentheses, otherwise it would look like `map` had four arguments. It can have any type $\alpha \to \beta$. That is to say, it can have any argument and result types, and they do not have to be the same as each other – though they may be. The argument has type α **list** because each of its elements must be an appropriate argument for `f`. In the same way, the result list has type β **list** because each of its elements is a result from `f` (in our `halve` example, α and β were both **int**). We can rewrite our `evens` function to use `map`:

```
is_even : int → bool
evens : int list → bool list

let is_even x =
  x mod 2 = 0

let evens l =
  map is_even l
```

In `evens`' use of `map`, α was **int**, β was **bool**. We can make `evens` still shorter: when we are just using a function once, we can define it directly, without naming it:

```
evens : int list → bool list

let evens l =
  map (fun x -> x mod 2 = 0) l
```

This is called an *anonymous function*. It is defined using **`fun`**, a named argument, the `->` arrow and the function definition (body) itself. For example, we can write our halving function like this:

```
fun x -> x / 2
```

and, thus, write:

$$\begin{array}{ll} & \underline{\texttt{map (fun x -> x / 2) [10; 20; 30]}} \\ \overset{*}{\Longrightarrow} & \texttt{[5; 10; 15]} \end{array}$$

We use anonymous functions when a function is only used in one place and is relatively short, to avoid defining it separately.

In the preceding chapter we wrote a sorting function and, in one of the questions, you were asked to change the function to use a different comparison operator so that the function would sort elements into reverse order. Now, we know how to write a version of the `msort` function which uses any comparison function we give it. A comparison function would have type $\alpha \rightarrow \alpha \rightarrow$ **bool**. That is, it takes two elements of the same type, and returns `true` if the first is "greater" than the second, for some definition of "greater" – or `false` otherwise.

So, let us alter our `merge` and `msort` functions to take an extra argument – the comparison function. The result is shown in Figure 6.1. Now, if we make our own comparison operator:

```
greater : α → α → bool

let greater a b =
  a >= b
```

```
merge : (α → α → bool) → α list → α list → α list
msort : (α → α → bool) → α list → α list

let rec merge cmp x y =
  match x, y with
    [], l -> l
  | l, [] -> l
  | hx::tx, hy::ty ->
      if cmp hx hy                                  use our comparison function
        then hx :: merge cmp tx (hy :: ty)          put hx first – it is “smaller”
        else hy :: merge cmp (hx :: tx) ty          otherwise put hy first

let rec msort cmp l =
  match l with
    [] -> []
  | [x] -> [x]
  | _ ->
      let left = take (length l / 2) l in
        let right = drop (length l / 2) l in
          merge cmp (msort cmp left) (msort cmp right)
```

Figure 6.1: Adding an extra argument to merge sort

we can use it with our new version of the `msort` function:

$\underline{\texttt{msort greater [5; 4; 6; 2; 1]}}$

$\overset{*}{\Longrightarrow}$ `[6; 5; 4; 2; 1]`

In fact, we can ask OCaml to make such a function from an operator such as <= or + just by enclosing it in parentheses and spaces:

```
        OCaml

# ( <= )
- : 'a -> 'a -> bool = <fun>
# ( <= ) 4 5
- : bool = true
```

So, for example:

$\underline{\texttt{msort (<=) [5; 4; 6; 2; 1]}}$

$\overset{*}{\Longrightarrow}$ `[1; 2; 4; 5; 6]`

and

$\underline{\texttt{msort (>=) [5; 4; 6; 2; 1]}}$

$\overset{*}{\Longrightarrow}$ `[6; 5; 4; 2; 1]`

The techniques we have seen in this chapter are forms of *program reuse*, which is fundamental to writing manageable large programs.

Questions

1. Write a simple recursive function `calm` to replace exclamation marks in a **char list** with periods. For example `calm ['H'; 'e'; 'l'; 'p'; '!'; ' '; 'F'; 'i'; 'r'; 'e'; '!']` should evaluate to `calm ['H'; 'e'; 'l'; 'p'; '.'; ' '; 'F'; 'i'; 'r'; 'e'; '.']`. Now rewrite your function to use `map` instead of recursion. What are the types of your functions?
2. Write a function `clip` which, given an integer, clips it to the range $1 \ldots 10$ so that integers bigger than 10 round down to 10, and those smaller than 1 round up to 1. Write another function `cliplist` which uses this first function together with `map` to apply this clipping to a whole list of integers.
3. Express your function `cliplist` again, this time using an anonymous function instead of `clip`.
4. Write a function `apply` which, given another function, a number of times to apply it, and an initial argument for the function, will return the cumulative effect of repeatedly applying the function. For instance, `apply f 6 4` should return `f (f (f (f (f (f 4))))))`. What is the type of your function?
5. Modify the insertion sort function from the preceding chapter to take a comparison function, in the same way that we modified merge sort in this chapter. What is its type?
6. Write a function `filter` which takes a function of type $\alpha \rightarrow$ **bool** and an α **list** and returns a list of just those elements of the argument list for which the given function returns `true`.
7. Write the function `for_all` which, given a function of type $\alpha \rightarrow$ **bool** and an argument list of type α **list** evaluates to `true` if and only if the function returns `true` for every element of the list. Give examples of its use.
8. Write a function `mapl` which maps a function of type $\alpha \rightarrow \beta$ over a list of type α **list list** to produce a list of type β **list list**.

So Far

1 Integers `min_int` … `-3 -2 -1 0 1 2 3` … `max_int` of type **int**. Booleans `true` and `false` of type **bool**. Characters of type **char** like `'X'` and `'!'`.

Mathematical operators `+ - * / mod` which take two integers and give another.

Operators `= < <= > >= <>` which compare two values and evaluate to either `true` or `false`.

The conditional **`if`** *expression1* **`then`** *expression2* **`else`** *expression3*, where *expresssion1* has type **bool** and *expression2* and *expression3* have the same type as one another.

The boolean operators `&&` and `||` which allow us to build compound boolean expressions.

2 Assigning a name to the result of evaluating an expression using the **`let`** *name* = *expression* construct. Building compound expressions using **`let`** *name1* = *expression1* **`in`** **`let`** *name2* = *expression2* **`in`** …

Functions, introduced by **`let`** *name argument1 argument2* … = *expression*. These have type $\alpha \rightarrow \beta$, $\alpha \rightarrow \beta \rightarrow \gamma$ etc. for some types α, β, γ etc.

Recursive functions, which are introduced in the same way, but using **`let rec`** instead of **`let`**.

3 Matching patterns using **`match`** *expression1* **`with`** *pattern1* `|` … `->` *expression2* `|` *pattern2* `|` … `->` *expression3* `|`… The expressions *expression2*, *expression3* etc. must have the same type as one another, and this is the type of the whole **`match`** … **`with`** expression.

4 Lists, which are ordered collections of zero or more elements of like type. They are written between square brackets, with elements separated by semicolons e.g. `[1; 2; 3; 4; 5]`. If a list is non-empty, it has a head, which is its first element, and a tail, which is the list composed of the rest of the elements.

The `::` "cons" operator, which adds an element to the front of a list. The `@` "append" operator, which concatenates two lists together.

Lists and the `::` "cons" symbol may be used for pattern matching to distinguish lists of length zero, one, etc. and with particular contents.

5 Matching two or more things at once, using commas to separate as in **`match`** `a, b` **`with`** `0, 0 ->` *expression1* `| x, y ->` *expression2* `|` …

6 Anonymous functions **`fun`** *name* `->` *expression*. Making operators into functions as in `( < )` and `( + )`.