Chapter 3

Basic concepts

We examine in this chapter some fundamental concepts which we will use and study in the following chapters. Some of them are specific to the interface with the Caml language (toplevel, global definitions) while others are applicable to any functional language.

3.1 Toplevel loop

The first contact with Caml is through its interactive aspect[1]. When running Caml on a computer, we enter a *toplevel loop* where Caml waits for input from the user, and then gives a response to what has been entered.

The beginning of a Caml Light session looks like this (assuming `$` is the shell prompt on the host machine):

```
$camllight
>       Caml Light version 0.6

#
```

On the PC version, the command is called `caml`.

The "`#`" character is Caml's prompt. When this character appears at the beginning of a line in an actual Caml Light session, the toplevel loop is waiting for a new toplevel phrase to be entered.

Throughout this document, the phrases starting by the `#` character represent legal input to Caml. Since this document has been pre-processed by Caml Light and these lines have been effectively given as input to a Caml Light process, the reader may reproduce by him/herself the session contained in each chapter (each chapter of the first two parts contains a different session, the third part is a single session). Lines starting with the "`>`" character are Caml Light error messages. Lines starting by another character are either Caml responses or (possibly) illegal input. The input is printed in typewriter font (`like this`), and output is written using slanted typewriter font (*like that*).

Important: Of course, when developing non-trivial programs, it is preferable to edit programs in files and then to include the files in the toplevel, instead of entering the programs directly.

[1]Caml Light implementations also possess a batch compiler usable to compile files and produce standalone applications; this will be discussed in chapter 11.

Furthermore, when debugging, it is very useful to *trace* some functions, to see with what arguments they are called, and what result they return. See the reference manual [20] for a description of these facilities.

3.2 Evaluation: from expressions to values

Let us enter an arithmetic expression and see what is Caml's response:

```
#1+2;;
- : int = 3
```

The expression "`1+2`" has been entered, followed by "`;;`" which represents the end of the current toplevel phrase. When encountering "`;;`", Caml enters the type-checking (more precisely *type synthesis*) phase, and prints the inferred type for the expression. Then, it compiles code for the expression, executes it and, finally, prints the result.

In the previous example, the result of evaluation is printed as "`3`" and the type is "`int`": the type of integers.

The process of evaluation of a Caml expression can be seen as transforming this expression until no further transformation is allowed. These transformations must preserve semantics. For example, if the expression "`1+2`" has the mathematical object 3 as semantics, then the result "`3`" must have the same semantics. The different phases of the Caml evaluation process are:

- parsing (checking the syntactic legality of input);
- type synthesis;
- compiling;
- loading;
- executing;
- printing the result of execution.

Let us consider another example: the application of the successor function to `2+3`. The expression `(function x -> x+1)` should be read as "the function that, given `x`, returns `x+1`". The juxtaposition of this expression to `(2+3)` is *function application*.

```
#(function x -> x+1) (2+3);;
- : int = 6
```

There are several ways to reduce that value before obtaining the result `6`. Here are two of them (the expression being reduced at each step is underlined):

$$\begin{tikzcd}
\texttt{(function x -> x+1) }\underline{\texttt{(2+3)}} \arrow[d] & \underline{\texttt{(function x -> x+1) (2+3)}} \arrow[d] \\
\underline{\texttt{(function x -> x+1) 5}} \arrow[d] & \underline{\texttt{(2+3)}}\texttt{ + 1} \arrow[d] \\
\underline{\texttt{5+1}} \arrow[d] & \underline{\texttt{5+1}} \arrow[d] \\
\texttt{6} & \texttt{6}
\end{tikzcd}$$

The transformations used by Caml during evaluation cannot be described in this chapter, since they imply knowledge about compilation of Caml programs and machine representation of Caml values. However, since the basic control structure of Caml is function application, it is quite easy to give an idea of the transformations involved in the Caml evaluation process by using the kind of rewriting we used in the last example. The evaluation of the (well-typed) application $e_1(e_2)$, where e_1 and e_2 denote arbitrary expressions, consists in the following steps:

- Evaluate e_2, obtaining its value v.
- Evaluate e_1 until it becomes a functional value. Because of the well-typing hypothesis, e_1 must represent a function from some type t_1 to some t_2, and the type of v is t_1. We will write (`function x ->` e) for the result of the evaluation of e_1. It denotes the mathematical object usually written as:

  $$\begin{array}{ll} f: & t_1 \rightarrow t_2 \\ & x \mapsto e \text{ (where, of course, } e \text{ may depend on } x) \end{array}$$

  N.B.: We do not evaluate e before we know the value of x.
- Evaluate e where v has been substituted for all occurrences of `x`. We then get the final value of the original expression. The result is of type t_2.

In the previous example, we evaluate:

- `2+3` to `5`;
- `(function x -> x+1)` to itself (it is already a function body);
- `x+1` where `5` is substituted for `x`, i.e. evaluate `5+1`, getting `6` as result.

It should be noticed that Caml uses call-by-value: arguments are evaluated before being passed to functions. The relative evaluation order of the functional expression and of the argument expression is implementation-dependent, and should not be relied upon. The Caml Light implementation evaluates arguments before functions (right-to-left evaluation), as shown above; the original Caml implementation evaluates functions before arguments (left-to-right evaluation).

3.3 Types

Types and their checking/synthesis are crucial to functional programming: they provide an indication about the correctness of programs.

The universe of Caml values is partitioned into *types*. A type represents a collection of values. For example, `int` is the type of integer numbers, and `float` is the type of floating-point numbers. Truth values belong to the `bool` type. Character strings belong to the `string` type. Types are divided into two classes:

- Basic types (`int`, `bool`, `string`, ...).

- Compound types such as functional types. For example, the type of functions from integers to integers is denoted by `int -> int`. The type of functions from boolean values to character strings is written `bool -> string`. The type of pairs whose first component is an integer and whose second component is a boolean value is written `int * bool`.

In fact, any combination of the types above (and even more!) is possible. This could be written as:

```
BasicType  ::=  int
             |  bool
             |  string

Type       ::=  BasicType
             |  Type -> Type
             |  Type * Type
```

Once a Caml toplevel phrase has been entered in the computer, the Caml process starts analyzing it. First of all, it performs *syntax analysis*, which consists in checking whether the phrase belongs to the language or not. For example, here is a syntax error[2] (a parenthesis is missing):

```
#(function x -> x+1 (2+3);;
Toplevel input:
>(function x -> x+1 (2+3);;
>                       ^^
Syntax error.
```

The carets "`^^^`" underline the location where the error was detected.

The second analysis performed is *type analysis*: the system attempts to assign a type to each subexpression of the phrase, and to synthesize the type of the whole phrase. If type analysis fails (i.e. if the system is unable to assign a sensible type to the phrase), then a type error is reported and Caml waits for another input, rejecting the current phrase. Here is a type error (cannot add a number to a boolean):

```
#(function x -> x+1) (2+true);;
Toplevel input:
>(function x -> x+1) (2+true);;
>                       ^^^^
This expression has type bool,
but is used with type int.
```

The rejection of ill-typed phrases is called *strong typing*. Compilers for weakly typed languages (C, for example) would instead issue a warning message and continue their work at the risk of getting a "`Bus error`" or "`Illegal instruction`" message at run-time.

Once a sensible type has been deduced for the expression, then the evaluation process (compilation, loading and execution) may begin.

Strong typing enforces writing clear and well-structured programs. Moreover, it adds security and increases the speed of program development, since most typos and many conceptual errors are

[2]Actually, lexical analysis takes place before syntax analysis and *lexical errors* may occur, detecting for instance badly formed character constants.

trapped and signaled by the type analysis. Finally, well-typed programs do not need dynamic type tests (the addition function does not need to test whether or not its arguments are numbers), hence static type analysis allows for more efficient machine code.

3.4 Functions

The most important kind of values in functional programming are functional values. Mathematically, a function f of type $A \rightarrow B$ is a rule of correspondence which associates with each element of type A a unique member of type B.

If x denotes an element of A, then we will write $f(x)$ for the application of f to x. Parentheses are often useless (they are used only for grouping subexpressions), so we could also write $(f(x))$ as well as $(f((x)))$ or simply $f\ x$. The value of $f\ x$ is the unique element of B associated with x by the rule of correspondence for f.

The notation $f(x)$ is the one normally employed in mathematics to denote functional application. However, we shall be careful not to confuse a function with its application. We say “the function f with formal parameter x”, meaning that f has been defined by:

$$f : x \mapsto e$$

or, in Caml, that the body of f is something like `(function x -> ...)`. Functions are values as other values. In particular, functions may be passed as arguments to other functions, and/or returned as result. For example, we could write:

```
#function f -> (function x -> (f(x+1) - 1));;
- : (int -> int) -> int -> int = <fun>
```

That function takes as parameter a function (let us call it `f`) and returns another function which, when given an argument (let us call it `x`), will return the predecessor of the value of the application `f(x+1)`.

The type of that function should be read as: `(int -> int) -> (int -> int)`.

3.5 Definitions

It is important to give names to values. We have already seen some named values: we called them *formal parameters*. In the expression `(function x -> x+1)`, the name `x` is a formal parameter. Its name is irrelevant: changing it into another one does not change the meaning of the expression. We could have written that function `(function y -> y+1)`.

If now we apply this function to, say, `1+2`, we will evaluate the expression `y+1` where `y` is the value of `1+2`. Naming `y` the value of `1+2` in `y+1` is written as:

```
#let y=1+2 in y+1;;
- : int = 4
```

This expression is a legal Caml phrase, and the `let` construct is indeed widely used in Caml programs.

The `let` construct introduces *local bindings of values to identifiers*. They are *local* because the scope of `y` is restricted to the expression `(y+1)`. The identifier `y` kept its previous binding (if any) after the evaluation of the "`let` ... `in` ..." construct. We can check that `y` has not been globally defined by trying to evaluate it:

```
#y;;
Toplevel input:
>y;;
>^
The value identifier y is unbound.
```

Local bindings using `let` also introduce *sharing* of (possibly time-consuming) evaluations. When evaluating "`let x=`e_1 `in` e_2", e_1 gets evaluated only once. All occurrences of `x` in e_2 access the *value* of e_1 which has been computed once. For example, the computation of:

```
let big = sum_of_prime_factors 35461243
in big+(2+big)-(4*(big+1));;
```

will be less expensive than:

```
  (sum_of_prime_factors 35461243)
+ (2 + (sum_of_prime_factors 35461243))
- (4 * (sum_of_prime_factors 35461243 + 1));;
```

The `let` construct also has a global form for toplevel declarations, as in:

```
#let successor = function x -> x+1;;
successor : int -> int = <fun>

#let square = function x -> x*x;;
square : int -> int = <fun>
```

When a value has been declared at toplevel, it is of course available during the rest of the session.

```
#square(successor 3);;
- : int = 16

#square;;
- : int -> int = <fun>
```

When declaring a functional value, there are some alternative syntaxes available. For example we could have declared the `square` function by:

```
#let square x = x*x;;
square : int -> int = <fun>
```

or (closer to the mathematical notation) by:

```
#let square (x) = x*x;;
square : int -> int = <fun>
```

All these definitions are equivalent.

3.6 Partial applications

A *partial application* is the application of a function to some but not all of its arguments. Consider, for example, the function `f` defined by:

```
#let f x = function y -> 2*x+y;;
f : int -> int -> int = <fun>
```

Now, the expression `f(3)` denotes the function which when given an argument `y` returns the value of `2*3+y`. The application `f(x)` is called a *partial application*, since `f` waits for two successive arguments, and is applied to only one. The value of `f(x)` is still a function.

We may thus define an addition function by:

```
#let addition x = function y -> x+y;;
addition : int -> int -> int = <fun>
```

This can also be written:

```
#let addition x y = x+y;;
addition : int -> int -> int = <fun>
```

We can then define the successor function by:

```
#let successor = addition 1;;
successor : int -> int = <fun>
```

Now, we may use our `successor` function:

```
#successor (successor 1);;
- : int = 3
```

Exercises

Exercise 3.1 *Give examples of functions with the following types:*

1. `(int -> int) -> int`
2. `int -> (int -> int)`
3. `(int -> int) -> (int -> int)`

Exercise 3.2 *We have seen that the names of formal parameters are meaningless. It is then possible to rename* `x` *by* `y` *in* `(function x -> x+x)`*. What should we do in order to rename* `x` *in* `y` *in*

```
(function x -> (function y -> x+y))
```

Hint: rename `y` *by* `z` *first. Question: why?*

Exercise 3.3 *Evaluate the following expressions (rewrite them until no longer possible):*

```
let x=1+2 in ((function y -> y+x) x);;
let x=1+2 in ((function x -> x+x) x);;
let f1 = function f2 -> (function x -> f2 x)
in let g = function x -> x+1
   in f1 g 2;;
```