Functional programming using CAML

Michel Mauny

Functional Programming using CAML[1]

Michel Mauny

Domaine de Voluceau
BP 105, F-78153 Le Chesnay Cedex
E-mail: Michel.Mauny@inria.fr

September 4, 1991.

[1]This is a slightly revised version of INRIA Technical Report 129 (May 1991).

Programmation fonctionnelle en CAML

Michel Mauny

Résumé

Ces notes de cours forment une introduction à la programmation fonctionnelle. Elles ont été rédigées lors d'un enseignement dispensé à l'Université de Salerne (Italie) au printemps 1990. Elles peuvent servir de support à un enseignement de programmation fonctionnelle à des étudiants ayant eu un premier contact avec la programmatiom. Cependant, aucune connaissance préalable n'est strictement requise.

A travers l'étude du langage fonctionnel CAML, nous présentons la synthèse de types polymorphes mise en œuvre par l'algorithme de Milner. Nous présentons aussi un modèle simple d'éxécution des langages fonctionnels: une version simplifiée de la Machine Abstraite Catégorique (CAM). Ces notes se terminent par le prototypage complet d'un petit langage fonctionnel.

Functional Programming using CAML

Michel Mauny

Abstract

These course notes represent an introduction to functional programming. They have been written while the author was teaching at the University of Salerno (Italy) during spring 1990. These notes can be used for teaching functional programming to students who already had a first contact with Computer Science. However, no prerequesite is strictly required.

Through the study of the CAML functional programming, we present polymorphic type synthesis implemented by Milner's algorithm. We also present a simple execution model for functional languages: a simplified version of the Categorical Abstract Machine (CAM). These notes end by the complete prototyping of a simple functional language.

Contents

Chapter 1

Introduction

These course notes are an introduction to functional programming. The course was first given at Salerno University (Italy) in spring 1990.

Through the study of the CAML ([Cousineau and Huet, 1990], [Weis *et al.*, 1990]) functional language, we present static type checking, polymorphism and type synthesis as implemented by Milner's algorithm ([Milner, 1978]). We also present a simple stack execution model of functional languages: a simplified version of the Categorical Abstract Machine (CAM) ([Cousineau *et al.*, 1985], [Mauny and Suárez, 1986]).

Functionality and type systems of functional languages such as CAML allow rapid implementation of prototypes: the programmer may define his/her own data structures as well as functions manipulating them with the security provided by a strict type verification.

After generalities concerning functional languages (part 1), we present some features of the CAML programming language, without being exhaustive (part 2). The third part is dedicated to a prototype implementation of a small and simple functional programming language.

These notes certainly contain imperfections. Suggestions and "bug reports" should be sent (preferably) by electronic mail to `Michel.Mauny@inria.fr`.

Warning: The programs and remarks (especially contained in parts 2 and 3) of this document might not be valid in CAML versions anterior or posterior to CAML 2-6.1.

Acknowledgements

I thank Giancarlo Nota for his invitation in Salerno and for making my stay there as nice as possible.

I must thank Daniel de Rauglaudre whose attentive reading of these notes led to several improvements. Thanks are also due to Amy Felty for correcting my English and to Thérèse Hardin and Didier Rémy for reading a previous version of these notes.

Part I

Functional Programming

Chapter 2

Functional languages

Programming languages are said to be *functional* when their basic component is the notion of *function* and their essential control structure is *function application*. For example, the Lisp [MacCarthy, 1962] language has been called functional because it possesses these two properties.

However, we want the programming notion of function to be as close as possible to the usual mathematical notion of function. In mathematics, functions are "first-class" objects: they can be arbitrarily manipulated. For example, they can be composed, and the composition function is itself a function.

In mathematics, one would present the *successor* function in the following way:

$$\begin{array}{rll} \mathit{successor}: & \mathbb{N} & \longrightarrow \mathbb{N} \\ & n & \longmapsto n+1 \end{array}$$

The functional composition could be presented as:

$$\begin{array}{rll} o: & (A \Rightarrow B) \times (C \Rightarrow A) & \longrightarrow (C \Rightarrow B) \\ & (f,g) & \longmapsto (x \longmapsto f\,(g\,x)) \end{array}$$

where $(A \Rightarrow B)$ denotes the space of functions from A to B.

We remark here the importance of:

1. the notion of *type*; a mathematical function always possesses a *domain* and a *codomain*. They will correspond to the programming notion of type.

2. lexical binding: when we wrote the mathematical definition of *successor*, we have assumed that the addition function + had been previously defined, mapping a pair of natural numbers to a natural number.

3. the notion of *functional abstraction*: the name *successor* is now used in order to represent the functional value mapping any integer n to the integer resulting of the computation of $n+1$. We would break common mathematical rules if we were allowed to change that text.

Now, if we take a look at the corresponding notions in the Lisp language[1], we can see that Lisp respects none of them.

[1] We are here talking about the traditional definition of the Lisp language: some modern versions of Lisp (e.g. Scheme) may be seen as "almost functional" (they still do not possess a type system, and their toplevel binding rule is still dynamic).

ML dialects (cf. below) respect these notions. But they also allow non-functional programming styles, and, in this sense, they are functional but not *purely functional.*

2.1 History of functional languages

Some historical points:

- 1930: Alonzo Church developed the λ-calculus [Church, 1941] as an attempt to provide a basis for mathematics. The λ-calculus is a formal theory for functionality. The three basic constructs of the λ-calculus are:
  - variable names (e.g. x, y,...);
  - application (MN if M and M are terms);
  - functional abstraction ($\lambda x.M$).

  Terms of the λ-calculus represent functions. The pure λ-calculus has been proved inconsistent as a logical theory. Some *type systems* have been added to it in order to remedy this inconsistency.
- 1958: Mac Carthy invented Lisp [MacCarthy, 1962] whose programs have some similarities with terms of λ-calculus. Lisp dialects have been recently evolving in order to be closer to modern functional languages (Scheme), but they still do not possess a type system.
- 1965: P. Landin proposed the ISWIM [Landin, 1966] language (for "If You See What I Mean"), which is the precursor of languages of the ML family.
- 1978: J. Backus introduced FP: a language of combinators [Backus, 1978] and a framework in which it is possible to reason about programs. The main particularity of FP is that there are no variable names in FP programs.
- 1978: R. Milner proposes a language called ML [Gordon *et al.*, 1978], intended to be the *metalanguage* of the LCF proof assistant (i.e. the language used to program the search of proofs). This language is inspired by ISWIM (close to λ-calculus) and possesses an original type system.
- 1985: D. Turner proposed the Miranda [Turner, 1985] programming language, which uses Milner's type system but where programs are submitted to *lazy evaluation.*

Currently, the two main families of functional languages are the ML and the Miranda families.

2.2 The ML family

ML languages are based on a sugared[2] version of λ-calculus. Their evaluation regime is *call-by-value* (i.e. the argument is evaluated before being passed to a function), and they use Milner's type system.

[2] i.e. with a more user-friendly syntax.

The LCF proof system appeared in 1972 in Stanford (Stanford LCF). It was further developed in Cambridge (Cambridge LCF) where the ML language was added to it.

From 1981 to 1986, a version of ML and its compiler was developed in a collaboration between INRIA and Cambridge by G. Cousineau, G. Huet and L. Paulson.

In 1981, L. Cardelli implemented a version of ML whose compiler generated native machine code.

In 1984, a commitee of researchers from the universities of Edinburgh and Cambridge, Bell Laboratories and INRIA, headed by R. Milner worked on a new extended language called Standard ML ([Milner, 1987]). This core language was completed by a module facility designed by D. MacQueen ([MacQueen, 1984]).

Since 1984, the CAML language has been under design in a collaboration between INRIA and LIENS[3]). Its first release appeared in 1987 and the main implementors of CAML were Ascánder Suárez, Pierre Weis and the author.

In 1989 appeared Standard ML of New-Jersey, developed by Andrew Appel (Princeton University) and David MacQueen (Bell Laboratories).

2.3 The Miranda family

All languages in this family use *lazy evaluation* (i.e. the argument of a function is evaluated if and when the function needs its value—arguments are passed unevaluated to functions). They also use Milner's type system.

Languages belonging to the Miranda family find their origin in the SASL [Turner, 1976] (1976) language developed by D. Turner. SASL and its successors (KRC [Turner, 1982] 1981, Miranda [Turner, 1985] 1985 and Haskell [Hudak and Wadler, 1990] 1990) use *sets of mutually recursive equations* as programs. These equations are written in a *script* (collection of declarations) and the user may evaluate expressions using values defined in this script. LML (Lazy ML) has been developed in Göteborg (Sweeden); its syntax is close to ML's syntax and it uses a fast execution model: the G-machine [Johnsson, 1984].

[3]Laboratoire d'Informatique de l'Ecole Normale Supérieure, 45 Rue d'Ulm, 75005 Paris

Chapter 3

Fundamental 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. 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 session looks like this (suppose **%** is the shell's prompt on the host machine):

```
%caml
   CAML (sony) (V 2-6.1) by INRIA Fri Nov 24 1989

#
```

The "**#**" character is CAML's prompt. Throughout this document, the lines starting by the **#** character represent legal input to CAML. Since this document has been pre-processed by CAML and these lines have been effectively given as input to a CAML process, the reader may reproduce by him/herself the session contained in each chapter (each chapter of the two first parts contains a different session, the third part is a single session). Lines not starting by the **#** character are either CAML responses or (possibly) illegal input.

3.2 Evaluation: from expressions to values

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

```
#1+2;;
3 : num
```

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, then compiles code for the expression, executes it and, finally, prints the result together with the type information previously computed. In the previous example, the result of evaluation is printed as "`3`" and the type is "`num`": the type of numbers.

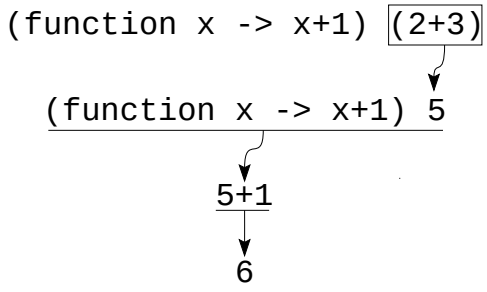
The process of evaluation of a CAML expression can be seen as transforming this expression until no further transformation is allowed. These transformations must preserve semantics. For example, if the expression "`1+2`" has the mathematical object 3 as semantics, then the result "`3`" must have the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

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

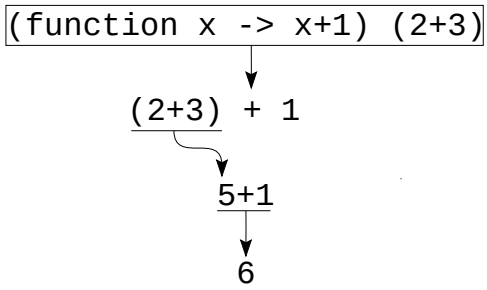
Let us consider another example: the application of the successor function to `2+3`:

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

There are several ways to reduce that value before obtaining the result `6`. For example, we could obtain successively:



or:



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_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). We will write (`function x ->` e) the result of evaluation of e_1. It denotes the mathematical object usually written as:

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

  N.B.: We do not evaluate e before we know the value of x.

- Evaluate e_2, obtaining its value v (because the program is well-typed, the type of v is t_1).

- 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:

- `(function x -> x+1)` to itself (it is already a function body);

- `2+3` to `5`;

- `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), and evaluates functions before arguments (i.e. CAML uses a left-to-right call-by-value evaluation strategy).

3.3 Types

Types and their checking/synthesis are crucial in functional programming. They provide an indication about correctness of programs.

The universe of CAML values is partitioned into *types*. A type represents a collection of values. For example, the type of numbers, denoted by `num`, contains mathematical integers, rational numbers and floating point numbers. Truth values belong to the `bool` type, and character strings belong to the `string` type.Types are divided into two classes:

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

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

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

```
BasicType  ::=  "num"
            |   "bool"
            |   "string"

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

Once a CAML toplevel phrase has been entered in the computer, the CAML process starts analysing that phrase. First of all, it does *syntax analysis* which consists in checking whether the phrase belongs to the language. The second analysis which is performed is *type-analysis*: the system tries to assign a type to each subexpression of the current phrase, and to synthetize 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. For example, here is a syntax error (a parenthesis is missing):

```
#(function x -> x+1 (2+3);;

line 1 Syntax error:
Skipping: ;;
Parse Failed
```

Here is a type error (cannot add a number to a string):

```
#(function x -> x+1) (2+"Hello world!");;

line 1: ill-typed phrase, the constant "Hello world!"
of type string cannot be used with type instance num in
2+"Hello world!"
1 error in typechecking

Typecheck Failed
```

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, during execution, getting a "`Bus error`" or "`Illegal instruction`" message, followed by a `core dumped` to use with our favorite debugger!

Once a sensible type has been deduced for the expression, then the evaluation mechanism (i.e. 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 logical errors are trapped and signalled 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), static type-analysis provides 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. In some mathematics texts one often finds the phrase "the function $f(x)$". This is because in these texts, functions are not considered as values. If that was the case, one would write "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));;
<fun> : ((num -> num) -> num -> num)
```

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: `(num -> num) -> (num -> num)`.

3.5 Definitions

It is important to give names to values. We already saw some names: we called them *formal parameters*. In the expression `(function x -> x+1)`, the name `x` is irrelevant: changing that name 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 that function to, say, `1+2`, we will evaluate the expression `y+1` *where* `y` is the value of `1+2`. We could write that intermediate expression as:

```
y+1 where y=1+2;;
```

or:

```
let y=1+2 in y+1;;
```

These two previous expressions are legal CAML expressions, and the `let` and `where` are indeed widely used in CAML programs.

The `let` and `where` constructs represent *local bindings of values to identifiers*. They also introduce *sharing* of (possibly time-consuming) evaluations. For example, the computation of:

```
let big = sum_of_prime_factors 354612437647645734734
in big+(2+big)-(4*(big+1));;
```

will be less expensive than:

```
  (sum_of_prime_factors 354612437647645734734)
+ (2 + (sum_of_prime_factors 354612437647645734734))
- (4 * (sum_of_prime_factors 354612437647645734734+1));;
```

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

```
#let successor = function x -> x+1;;
Value successor = <fun> : (num -> num)

#let square = function x -> x*x;;
Value square = <fun> : (num -> num)
```

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

```
#square(successor 3);;
16 : num

#square;;
<fun> : (num -> num)
```

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;;
Value square = <fun> : (num -> num)
```

or (closer to the mathematical notation) by:

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

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;;
Value f = <fun> : (num -> num -> num)
```

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;;
Value addition = <fun> : (num -> num -> num)
```

which we could have written:

```
#let addition x y = x+y;;
Value addition = <fun> : (num -> num -> num)
```

And then define the successor function by:

```
#let successor = addition 1;;
Value successor = <fun> : (num -> num)
```

Now, we may use our `successor` function:

```
#successor (successor 1);;
3 : num
```

Exercises

3.1 Give examples of functions with the following types:

```
(num -> num) -> num
num -> (num -> num)
(num -> num) -> (num -> num)
```

3.2 We saw that the names of formal parameters were 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?

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;;
```

Chapter 4

Basic types

We examine in this chapter the CAML basic types.

4.1 Numbers

CAML numbers (the `num` type) include (arbitrarily big) integers, rational numbers and floating point numbers. Examples of such numbers are:

$$42 \quad -42 \quad 0 \quad 13.245 \quad 1/2 \quad -0.6e6 \quad 0.7e-2$$

Integers and rational numbers have an arbitrary precision (all computations are *exact*, but may sometimes be quite slow). Floating point numbers are of limited size, and thus some results may be surprisingly false because some overflow occurred during the computation.

Predefined operations (functions) on numbers are:

`+` addition
`-` subtraction and unary minus
`*` multiplication
`/` division
`quo` integer division
`mod` modulo

Some examples of expressions:

```
#2/4;;
1/2 : num

#2/4 * 4 + 2;;
4 : num

#2/4 * (4 + 2);;
3 : num

#3 - 7 - 2;;
-6 : num
```

Since all these operators have an infix status (they are written between their arguments), there must be a notion of binding power and precedence. See the CAML Reference Manual in order to get exact information about these precedences and binding powers. Otherwise, use parentheses in order to group expressions: if the programmer knows exactly all CAML parsing rules, the reader should not need to know them in order to understand a program.

So far, we have not seen the type of these arithmetic operations. All of them wait for a pair of arguments (cartesian product of `num` by `num`). The value of such infix identifiers may be obtained by taking their *prefix* version.

```
#prefix +;;
<fun> : (num * num -> num)

#prefix *;;
<fun> : (num * num -> num)

#prefix mod;;
<fun> : (num * num -> num)
```

We start by defining some very simple functions on numbers:

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

#square 2;;
4 : num

#square (2/3);;
4/9 : num

#let sum_of_squares (x,y) = square(x) + square(y);;
Value sum_of_squares = <fun> : (num * num -> num)

#let half_pi = 3.14159/2
#in sum_of_squares(cos(half_pi), sin(half_pi));;
1.0 : num
```

We now develop a classical example: the computation of the root of a function by Newton's method.

The statement of Newton's method says that if y is an approximation to a root of a function f, then:

$$y - \frac{f(y)}{f'(y)}$$

is a better approximation, where $f'(y)$ is the derivative of f evaluated at y. For example, with $f(x) = x^2 - a$, we obtain $f'(x) = 2x$ and so:

$$y - \frac{f(y)}{f'(y)} = y - \frac{y^2 - a}{2y} = \frac{y + \frac{a}{y}}{2}$$

We can define a function `deriv` for approximating the derivative of a function at a given point by:

```
#let deriv f x dx = (f(x+dx) - f(x))/dx;;
Value deriv = <fun> : ((num -> num) -> num -> num -> num)
```

Provided `dx` is sufficiently small, this gives a reasonable estimate of the derivative of f at x.

```
#let abs x = if x>0 then x else -x;;
Value abs = <fun> : (num -> num)

#let newton f epsilon = until satisfied improve
#  where rec until p change x =
#             if p(x) then x
#             else until p change (change(x))
#  and satisfied y = abs(f y) < epsilon
#  and improve y = y - (f(y) / (deriv f y epsilon));;
Value newton = <fun> : ((num -> num) -> num -> num -> num)
```

The `until` function is an example of a *recursive* function: it calls itself in its body. If `p(x)` is false, then `until` is called with new arguments. We will study recursive functions more closely later.

Let us now finish with our example:

```
#let square_root x epsilon =
#           newton (function y -> y*y -x) epsilon x
#and cubic_root x epsilon =
#           newton (function y -> y*y*y -x) epsilon x;;
Value square_root = <fun> : (num -> num -> num)
Value cubic_root = <fun> : (num -> num -> num)

#square_root 2 1e-5;;
1.414214 : num

#cubic_root 8 1e-5;;
2.0 : num

#2 * (newton cos 1e-5 1.5);;
3.141593 : num
```

4.2 Boolean values

The presence of the conditional construct implies the presence of boolean values. The type `bool` is composed of two values `true` and `false`.

```
#true;;
true : bool
```

```
#false;;
false : bool
```

The identifiers `true` and `false` are CAML keywords: they cannot be used as variable names. The functions with `bool` as codomain are often called *predicates*. Many predicates are predefined in CAML. Here is one of them:

```
#prefix <;;
<fun> : (num * num -> bool)
```

There exist also `prefix <=`, `prefix >`, `prefix >=`.

4.2.1 Equality

Equality has a special status in functional languages because of functional values. It is easy to test equality of boolean values:

```
#true = true;;
true : bool

#false = (1>2);;
true : bool
```

But it is impossible, in the general case, to decide the equality of functional values. Several choices are possible: to forbid equality testing of functional values; this can be realised by assigning the property of "admitting equality" to some types (SML, Haskell). In that case, functional types do not admit equality. The choice adopted in CAML was to allow equality testing on any type, but the user has to know that these tests are not safe on functional values (the answer may be `false` when testing equality of two equal functions). The usage of equality on functional values is thus not recommended.

Since equality may be used on values of any type, what is its type? Equality takes a pair of arguments of the same type (whatever type it is) and returns a boolean value. The type of equality is a *polymorphic type*, i.e. may take several possible forms. Indeed, when testing equality of two numbers, then its type is `num * num -> num`, and when testing equality of strings, its type is `string * string -> num`.

```
#prefix =;;
<fun> : ('a * 'a -> bool)

#1=2;;
false : bool

#(1,2) = (2,1);;
false : bool

#1 = (1,2);;

line 1: ill-typed phrase, 1,2 has an instance of type:
```

```
(num * num) which should match type: num in 1,2
1 error in typechecking

Typecheck Failed
```

4.2.2 Logical operators

The classical logical operators are available in CAML.

```
#true or false;;
true : bool

#(1<2) & (2>1);;
true : bool
```

The `not` operator excluded, they are not functions. They should be seen as macros, since, for example, the `or` operator evaluates its second operand only if the first one evaluates to `false`.

The behaviour of these operators may be described as follows:

e_1 `or` e_2 is equivalent to `if` e_1 `then true else` e_2

e_1 `&` e_2 is equivalent to `if` e_1 `then` e_2 `else false`

4.3 Strings

String constants are written between " characters (double-quotes). The most used string operation is string concatenation denoted by the ^ character.

```
#"Hello" ^ " World!";;
"Hello World!" : string

#prefix ^;;
<fun> : (string * string -> string)
```

Other operations are available (substring, etc). See the Reference manual for more information.

4.4 Pairs

4.4.1 Constructing pairs

It is possible to combine values into pairs. The *value constructor* of pairs is the "," character (the comma). We will often use parentheses around pairs in order to improve readability, but they are not strictly necessary.

```
#1,2;;
(1,2) : (num * num)

#let p = (1+2, 1<2);;
Value p = (3,true) : (num * bool)
```

The infix "*" identifier is the *type constructor* of pairs. It corresponds to the mathematical cartesian product.

We can build pairs from any values: the pair value constructor is *generic*.

4.4.2 Extracting pair components

Two *projection* functions are used in order to extract components of pairs:

```
#fst;;
<fun> : ('a * 'b -> 'a)

#snd;;
<fun> : ('a * 'b -> 'b)
```

They have of course a polymorphic type since they may be applied to any kind of pair. They are predefined in the CAML system, but could be defined by the user (in fact, the cartesian product itself could be defined — see the chapter dedicated to user-defined product types):

```
#let first (x,y) = x
#and second (x,y) = y;;
Value first = <fun> : ('a * 'b -> 'a)
Value second = <fun> : ('a * 'b -> 'b)

#first p;;
3 : num

#second p;;
true : bool
```

We say that `first` is a *generic* value because it works uniformly on several kinds of arguments (provided they are pairs). We often confuse between "generic" and "polymorphic", saying that such value is polymorphic instead of generic.

4.5 Patterns and pattern-matching

Patterns and pattern-matching play an important role in ML languages. They appear in all real-size programs and are strongly related to types (predefined or user-defined).

A *pattern* indicates the *shape* of a value. Patterns are "values with holes". A single variable (formal parameter) is a pattern (with no shape specified: it matches any value). When a value is *matched against* a pattern (this is called *pattern-matching*), then the pattern acts as a filter. We already used patterns and pattern-matching in all the functions we wrote: the function body `(function x -> ...)` does (trivial) pattern-matching. When applied to a value, then the formal parameter `x` gets bound to that value. The function body `(function (x,y) -> x+y)` does also pattern-matching: when applied to a value (a pair, because of well-typing hypotheses), the `x` and `y` get bound respectively to the first and the second component of that pair.

All these pattern-matching examples were trivial ones: they did not involve any test:

- formal parameters such as `x` do not impose any particular shape to the value they are supposed to match;
- pair patterns such as `(x,y)` always match for typing reasons (cartesian product is a *product type*).

However, some types do not garantee such a uniform shape to their values. Consider the `bool` type: an element of type `bool` is either `true` or `false`.

If we impose to a value of type `bool` to have the shape of `true`, then pattern-matching may fail.

Consider the following function:

```
#let f = function true -> false;;
Warning: 1 partial match in this phrase
Value f = <fun> : (bool -> bool)
```

The compiler warns us that the pattern-matching may fail (we did not consider the `false` case).

Thus, if we apply `f` to a value which evaluates to `true`, pattern-matching will succeed (an equality test is performed during execution).

```
#f (1<2);;
false : bool
```

But, if `f`'s argument evaluates to `false`, a run-time error is reported:

```
#f (1>2);;

Pattern matching Failed
```

Here is a correct definition using pattern-matching:

```
#let negate = function true -> false
#                    | false -> true;;
Value negate = <fun> : (bool -> bool)
```

The pattern-matching has now two cases, separated by the "`|`" character.

Cases are examined in turn, from top to bottom. An equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                    | x -> true;;
Value negate = <fun> : (bool -> bool)
```

The second case now matches any boolean value (in fact, only `false` since `true` has been caught by the first match case). When the second case is chosen, then the identifier `x` gets bound to the argument of `negate`, and could be used in the right-hand part of the match case. Since, in our example, we do not use the value of the argument in the right-hand part of the second match case, another equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                    | _ -> true;;
Value negate = <fun> : (bool -> bool)
```

Where "_" acts as a formal paramenter (matches any value), but does not introduce any binding: it should be read as "anything else".

As an example of pattern-matching, consider the following function definition (truth value table of implication):

```
#let imply = function (true,true) -> true
#                   | (true,false) -> false
#                   | (false,true) -> true
#                   | (false,false) -> true;;
Value imply = <fun> : (bool * bool -> bool)
```

Here is another way of defining `imply`, by using variables:

```
#let imply = function (true,x) -> x
#                   | (false,x) -> true;;
Value imply = <fun> : (bool * bool -> bool)
```

Simpler, and simpler again:

```
#let imply = function (true,x) -> x
#                   | (false,_) -> true;;
Value imply = <fun> : (bool * bool -> bool)

#let imply = function (true,false) -> false
#                   | _ -> true;;
Value imply = <fun> : (bool * bool -> bool)
```

Pattern-matching is allowed on any type of value (non-trivial pattern-matching is only possible on types with *data constructors*).

For example:

```
#let is_zero = function 0 -> true | _ -> false;;
Value is_zero = <fun> : (num -> bool)

#let is_yes = function "oui" -> true
#                    | "si" -> true
#                    | "ya" -> true
#                    | "yes" -> true
#                    | _ -> false (* Unknown language *);;
Value is_yes = <fun> : (string -> bool)
```

4.6 Functions

The type constructor "`->`" is predefined and cannot be defined in ML's type system. We shall explore in this section some further aspects of functions and functional types.

4.6.1 Functional composition

Functional composition is easily definable in CAML. It is of course a polymorphic function:

```
#let compose (f,g) = function x -> f (g (x));;
Value compose = <fun> :
      (('a -> 'b) * ('c -> 'a) -> 'c -> 'b)
```

The type of `compose` contains no more constraints than the ones appearing in the definition: in a sense, it is the *most general* type compatible with these constraints.

These constraints are:

- the codomain of `g` and the domain of `f` must be the same;
- `x` must belong to the domain of `g`;
- `compose f,g x` will belong to the codomain of `f`.

Functional composition is in fact already defined in the CAML system and bound to the infix identifier `o`:

```
#prefix o;;
<fun> : (('a -> 'b) * ('c -> 'a) -> 'c -> 'b)

#let succ x = x+1;;
Value succ = <fun> : (num -> num)

#succ o length;;
<fun> : ('a list -> num)

#(succ o length) [1;2;3];;
4 : num
```

4.6.2 Currifycation

We saw previously two versions of the addition function:

```
#let plus = prefix +;;
Value plus = <fun> : (num * num -> num)

#let add = function x -> (function y -> x+y);;
Value add = <fun> : (num -> num -> num)
```

These two functions differ only in their way of taking their arguments. The first one will take an argument belonging to a cartesian product, the second one will take a number and return a function. The `add` function is said to be *the curried version* of `plus` (in honor of the logician Haskell Curry).

The curryfication transformation may be written in CAML under the form of a higher-order function:

```
#let curry f = function x -> (function y -> f(x,y));;
Value curry = <fun> : (('a * 'b -> 'c) -> 'a -> 'b -> 'c)
```

Its inverse function may be defined by:

```
#let uncurry f = function (x,y) -> f x y;;
Value uncurry = <fun> : (('a -> 'b -> 'c) -> 'a * 'b -> 'c)
```

We may check the types of `curry` and `uncurry` on particular examples:

```
#curry (prefix +);;
<fun> : (num -> num -> num)

#curry (prefix ^);;
<fun> : (string -> string -> string)

#uncurry add;;
<fun> : (num * num -> num)
```

Exercises

4.1 Define in CAML functions computing the surface/volume of well-known geometric objects.

4.2 What would happen in a language submitted to call-by-value with recursion if there was no conditional construct (`if ... then ... else ...`)?

4.3 Define the `factorial` function such that:

$$\texttt{factorial}\ n = n * (n-1) * \ldots * 2 * 1$$

Give two versions of `factorial`: recursive and terminal recursive.

4.4 Define the `fibinacci` function which, when given a number n, returns the nth Fibonnacci number, i.e. the nth term u_n of the sequence defined by:

$$\begin{aligned} u_1 &= 1 \\ u_2 &= 1 \\ u_{n+2} &= u_{n+1} + u_n \end{aligned}$$

4.5 What are the types of the following expressions?

- `curry (prefix o)`
- `curry o uncurry`
- `uncurry o curry`

Chapter 5

Lists

Lists represent an important data structure, mainly because of their success in the Lisp language.

Lists in ML are *homogeneous*: a list may not contain elements of different types. This may be annoying to new ML users, yet lists are not as fundamental as in Lisp, since ML provides a facility for introducing new types allowing the user to define more precisely the data structures needed by the program.

5.1 Building lists

Lists are empty or non empty sequences of elements. They are built with two *value constructors*:

- `[]`, the empty list (read *nil*);
- `::`, the **cons**tructor, taking an element e and a list l, and builds a new list whose first element (*head*) is e and rest (*tail*) is l.

We may build lists of numbers:

```
#1::2::[];;
[1; 2] : num list

#[3;4;5];;
[3; 4; 5] : num list

#let x=2 in [1; 2; x+1; x+2];;
[1; 2; 3; 4] : num list
```

Lists of functions:

```
#let adds =
#  let add x y = x+y
#  in [add 1; add 2; add 3];;
Value adds = [<fun>; <fun>; <fun>] : (num -> num) list
```

and indeed, lists of anything desired.

We may wonder what are the types of the list (data) constructors. The empty list is a list of anything (since it has no element), it has thus the type "*list of anything*". The list **cons**tructor takes an element and a list (containing elements with the same type) and returns a new list. Here again, there is no type constraint on the elements considered.

```
#[];;
[] : 'a list

#prefix ::;;
<fun> : ('a * 'a list -> 'a list)
```

We see here that the `list` type is a *recursive* type. The `prefix ::` constructor receives an argument pair whose second component is a `list`.

5.2 Extracting elements from lists: pattern-matching

We know how to build lists; we see now show how to test emptyness of lists (although the equality predicate could be used for that) and extract elements from lists (e.g the first one). We used pattern-matching on pairs, numbers or boolean values. We shall extend the syntax of patterns to list patterns (in fact, we will see that any data constructor can be used in a pattern). For lists, at least two cases have to be considered (empty, non empty):

```
#let is_null = function [] -> true | _ -> false;;
Value is_null = <fun> : ('a list -> bool)

#let head = function x::_ -> x
#                  | _ -> raise failure "head";;
Value head = <fun> : ('a list -> 'a)

#let tail = function _::L -> L
#                  | _ -> raise failure "tail";;
Value tail = <fun> : ('a list -> 'a list)
```

Expression `raise failure "head"` will produce run-time errors since we chose to forbid taking the head of an empty list. We could have chosen `tail []` to evaluate to `[]`, but we cannot return a value for `head []` without changing the type of the `head` function. We say that the `list` type is a *sum type*, as `bool`, for example, because it is defined with several alternatives:

- a list is either `[]` or `::` of ...
- a boolean value is `true` or `false`

are examples of sum types. Sum types are the only types whose values need run-time tests in order to be matched by a non-variable pattern (i.e. a pattern which is not a single variable).

The cartesian product is a *product* type (only one alternative). Product types do not involve run-time tests during pattern-matching, because the type of their values is sufficient to indicate statically what their structure is.

5.3 Functions over lists

We will see in this section the definition of some useful functions over lists. These functions are of general interest, but are not exhaustive: many others are predefined in the CAML system (see [Cousineau and Huet, 1990] or [Weis *et al.*, 1990]). The definitions given here do not produce the most efficient version of each function; these definitions only intend to be easy to read and easy to understand.

Computation of the length of a list:

```
#let rec length = function [] -> 0
#                        | _::L -> 1 + length(L);;
Value length = <fun> : ('a list -> num)

#length [true; false];;
2 : num
```

Appending two lists:

```
#let rec append =
#     function [],L2 -> L2
#            | (x::L1), L2 -> x::(append (L1,L2));;
Value append = <fun> : ('a list * 'a list -> 'a list)
```

The `append` function is already defined in CAML, and bound to the infix identifier `@`.

```
#[1;2]@[3;4];;
[1; 2; 3; 4] : num list
```

Reversing a list:

```
#let rec rev = function [] -> []
#                     | x::L -> (rev L)@[x];;
Value rev = <fun> : ('a list -> 'a list)

#rev [1;2;3];;
[3; 2; 1] : num list
```

The `map` function combines functionality (it takes a function as argument), list processing and polymorphism (note the sharing of type variables between the arguments of `map` in its type). It maps a function over a list, and returns the list of results:

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

#map (function x -> x+1) [1;2;3;4;5];;
[2; 3; 4; 5; 6] : num list

#map length [ [1;2;3]; [4;5]; [6]; [] ];;
[3; 2; 1; 0] : num list
```

The following function is a list iterator. It is a bit harder to understand in the general case, but its behaviour is quite simple when given a curried addition function, `0` as base element, and a list of numbers. Given a function f, a base element a and a list $[x_1;\ldots;x_n]$, we have:

$$\texttt{it\_list}\ f\ a\ [x_1;\ldots;x_n] = f\ (\ldots(f\ (f\ a\ x_1)\ x_2)\ \ldots)x_n$$

For example, we have:

```
it_list add 0 [1;2;3;4;5] = 15

#let rec it_list f a =
#        function [] -> a
#               | x::L -> it_list f (f a x) L;;
Value it_list = <fun> :
      (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a)

#let sigma = it_list add 0;;
Value sigma = <fun> : (num list -> num)

#sigma [1;2;3;4;5];;
15 : num

#it_list (function x -> function y -> x*y) 1
#        [1;2;3;4;5];;
120 : num
```

The `it_list` function represents one of the many ways to iterate over a list, for other list iteration functions, see [Cousineau and Huet, 1990].

Exercises

5.1 Define the `combine` function which, when given a pair of lists, returns a list of pairs such that:

```
combine ([x1;...;xn],[y1;...;yn]) = [(x1,y1);...;(xn,yn)]
```

We suppose that the argument lists have the same length.

5.2 Define a function which, when given a list, returns the list of all its sublists.

Chapter 6

User-defined types

The user is allowed to define his/her own data types. With this facility, one does not have to find clever encodings for the data structures that must be manipulated by a program. 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 already saw 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 exactly 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 labelled* products of types. They are a generalization of cartesian product (cartesian product realizes only the product of two types).

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");;
Value jean = ("Jean",23,"Student","Paris") :
      (string * num * string * string)
```

and use projections `fst` and `snd` 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 `fst` (and indeed would be the

same function). The type of that function would not represent the fact that it is supposed to be applied to values such as `jean`, and could be applied to any pair.

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

```
#type person =
#  {Name:string; Age:num; Job:string; City:string};;
Type person defined
     .Name : (person -> string)
   ; .Age : (person -> num)
   ; .Job : (person -> string)
   ; .City : (person -> string)
```

The type `person` is the *product* of `string`, `num`, `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 nested pairs.

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

That 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;;
Value age_of = <fun> : (person -> num)
```

Since some labels may be sufficient for the CAML type-checker to find the type of a record expression, we can write:

```
#let job_of = function {Job=j; _} -> j;;
Value job_of = <fun> : (person -> string)

#job_of jean;;
"Student" : string
```

```
#let city_of p = p.City;;
Value city_of = <fun> : (person -> string)

#match jean with {City=c;_} -> c;;
"Paris" : string
```

The "`match <expr> with <match cases>`" construct is used when we want to match the value of a particular expression (match `<expr>` against each case of `<match cases>`). It is indeed equivalent to the application of `(function <match cases>)` to `<expr>`).

We have to be aware of other product types with the same labels. A CAML predefined product type with a label `Name` already exists. This is why the type-checking of the following definition fails: the type-checker could not find a unique type for the formal parameter `p`. It had two possible types: the type `person` and the predefined CAML product type with a label `Name`.

```
#let name_of p = p.Name;;

line 1: ill-typed phrase,
cannot resolve overloading ambiguity in p
1 error in typechecking

Typecheck Failed
```

The label `Name` is said to be *overloaded*.

We may have to help the CAML type-checker to find the type of some expressions by providing it type information. Such type information could be provided in several ways. In the following example, the first way is to mention another label of the type `person` in order to specify a bit more the shape of the argument. The second way is to give a *type constraint* on the argument, giving directly the required type information.

```
#let name_of {Name=s; Age=n;_} = s;;
Value name_of = <fun> : (person -> string)

#let name_of (p:person) = p.Name;;
Value name_of = <fun> : (person -> string)
```

In `(p:person)`, we specified that the formal parameter `p` has type `person`.

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
     .Fst : (('a,'b) pair -> 'a)
```

```
   ; .Snd : (('b,'a) pair -> 'a)

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

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

#first(p);;
3 : num
```

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

```
#fst p;;

line 1: ill-typed phrase, the variable p of type
(num,bool) pair cannot be used with type instance
('a * 'b) in fst p
1 error in typechecking

Typecheck Failed
```

6.2 Sum types

A *sum* type is the *finite labelled* 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 numbers (social security number).

We then need a type containing *at the same time* numbers and character strings. We also want to preserve static type-checking, we thus want to be able to distinguish numbers 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 num;;
Type identification defined
     'Name : (string -> identification)
   | 'SS : (num -> identification)
```

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

We use the "`'`" character to distinguish injections from other values.

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

```
#let id1 = 'Name "Jean"
#and id2 = 'SS (1670728280305 + 1);;
Value id1 = ('Name "Jean") : identification
Value id2 = ('SS (1670728280306)) : identification
```

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

```
#[id1;id2];;
[('Name "Jean"); ('SS (1670728280306))] :
 identification list
```

Pascal *enumerated types* are nothing but a particular case of CAML sum types. An enumerated type can be seen as the finite, labelled disjoint union of a special type `unit` containing only one element.

The type *unit* could be defined as the empty product type:

```
type unit = {};;
```

It is already defined in the CAML system. The only value of type unit is written `()` or `{}`.

An example of enumerated type is:

```
#type suit = 'Heart of unit
#          | 'Diamond of unit
#          | 'Club of unit
#          | 'Spade of unit;;
Type suit defined
     'Heart : (unit -> suit)
   | 'Diamond : (unit -> suit)
   | 'Club : (unit -> suit)
   | 'Spade : (unit -> suit)

#'Club {};;
('Club ()) : suit
```

The type `suit` contains only 4 distinct elements. Since the type `unit` contains a single element, we could omit it in the type definition: the labels are sufficient in order to distinguish elements of `suit`.

Here is the canonical way of defining the type `suit`:

```
#type suit = 'Heart | 'Diamond | 'Club | 'Spade;;
Warning: type suit redefined
Type suit defined
     'Heart : suit
   | 'Diamond : suit
```

```
   | 'Club : suit
   | 'Spade : suit

#'Club;;
'Club : suit
```

in which we consider injections as constants.

Let us continue this example by defining a type for cards.

```
#type card = 'Ace of suit
#          | 'King of suit
#          | 'Queen of suit
#          | 'Jack of suit
#          | 'Plain of (suit * num);;
Type card defined
     'Ace : (suit -> card)
   | 'King : (suit -> card)
   | 'Queen : (suit -> card)
   | 'Jack : (suit -> card)
   | 'Plain : (suit * num -> card)
```

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`,
- `num * 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];;
Value figures_of = <fun> : (suit -> card list)
Value small_cards_of = <fun> : (suit -> card list)

#figures_of 'Heart;;
[('Ace 'Heart); ('King 'Heart); ('Queen 'Heart);
 ('Jack 'Heart)] : card list

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

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";;
Value color = <fun> : (suit -> string)
```

Let us count the values of cards (assume we are playing at "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;;
Value count = <fun> : (suit -> card -> num)
```

6.2.3 Recursive types

Some types possess a naturally recursive structure (lists, for example).

Let us define a type for abstract syntax trees of a simple arithmetic language. An arithmetic expression will be either a numeric constant, or a variable, or the addition (multiplication, difference, and division) of two expressions.

```
#type arith_expr = 'Const of num
#                | '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
     'Const : (num -> arith_expr)
   | 'Var : (string -> arith_expr)
   | 'Plus : (args -> arith_expr)
   | 'Mult : (args -> arith_expr)
   | 'Minus : (args -> arith_expr)
   | 'Div : (args -> arith_expr)
Type args defined
     .Arg1 : (args -> arith_expr)
   ; .Arg2 : (args -> arith_expr)
```

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

Since abstract syntax trees are naturally recursive, their type definition is also recursive. Furthermore, the types `arith_expr` and `args` are defined in a mutually recursive way.

The recursive definition of types may lead to types such that it is impossible to build values of these types.

Example:

```
#type stupid = {Head:stupid; Tail:stupid};;
Type stupid defined
     .Head : (stupid -> stupid)
   ; .Tail : (stupid -> stupid)
```

Elements of this type are *infinite* data structures, and could be constructed in the following way:

```
#let rec stupid_value =
#    {Head=stupid_value; Tail=stupid_value};;

Ill construction of a recursive value
```

But the compiler refuses to accept this.

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. This is not necessary when using lazy evaluation (see section 7.1).

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
     'Empty : 'a sequence
   | 'Sequence : ('a * 'a sequence -> 'a sequence)
```

Indeed, in the *prelude* file of the CAML system, we may find the following definition:

```
type 'a list = [] | prefix :: of ('a * 'a list);;
```

defining the `list` type.

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
     'Leaf : ('a -> ('b,'a) btree)
   | 'Btree : (('a,'b) node -> ('a,'b) btree)
Type node defined
     .Op : (('a,'b) node -> 'a)
   ; .Son1 : (('a,'b) node -> ('a,'b) btree)
   ; .Son2 : (('a,'b) node -> ('a,'b) btree)

#let t = 'Btree {Op="+"; Son1='Leaf 1; Son2='Leaf 2};;
Value t =
      ('Btree {Op="+"; Son1=('Leaf 1); Son2=('Leaf 2)}) :
      (string,num) btree
```

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.

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. Other sum data constructors possess a functional type, so why don't we push the assimilation as far as possible. The reason is that 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;;
Value King = <fun> : (suit -> card)
```

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

```
#function 'King c -> c;;
Warning: 1 partial match in this phrase
<fun> : (card -> suit)
```

is `'King`$^{-1}$. It is a partial function, since pattern-matching may fail. When using the convention of injection names starting with the "`'`" character, we differentiate injections and their associated functions. This is why the following expression produces a type error:

```
#'King;;

line 1: illegal use of functional constructor 'King
as a constant constructor in 'King
1 error in typechecking

Typecheck Failed
```

However, it is possible to write a sum data type definition which defines at the same time constructors and their associated functions:

```
#type identification = Name of string | SS of num;;
Warning: type identification redefined
Type identification defined
     Name : (string -> identification)
   | SS : (num -> identification)

#Name;;
<fun> : (string -> identification)
```

In that case, `Name` stands for *both* the data constructor and for the function.

This feature is still present in CAML for reasons of compatibility with older versions of CAML. It is however a common source of errors. When data constructors have a syntax similar to other identifiers, then mispelling a constant data constructor in a pattern makes the CAML typechecker consider it as a formal parameter which matches any value.

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 num;;
Type counter1 defined
     'Counter : (num -> counter1)
```

Of course, the type `counter` is isomorphic to `num`. The injection `'Counter` is a *total* function from `num` to `counter1`. It is thus a *bijection*.

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

```
#type counter2 = {Counter: num};;
Type counter2 defined
     .Counter : (counter2 -> num)
```

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

The possibility of defining arbitrary complex data types permits the easy manipulation of abstract syntax trees in CAML (as the `arith_expr` example). 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 may be introduced in CAML.
- Types may be *parameterized* by type variables. The syntax of type parameters is:

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

- Types may 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>`.
  - We use the convention of choosing as names for `<Injection>` identifiers starting with the "`'`" character (e.g `'Plus`). This is only a convention: usual identifiers are also legal names.

Exercises

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

6.2 Define a function `map_btree` analoguous to the `list` map function. 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.

6.3 We can associate to the `list` type definition a canonical iterator by giving a functional interpretation to the data constructors of the `list` type (`[]` and `::`, that we will note respectively `nil` and prefix `cons`). More precisely, if given a list l, we abstract it over the list data constructors, we obtain:

```
(function cons -> (function nil -> l))
```
[1]

If now we abstract the previous expression over `l`, we obtain a general iterator over arbitrary lists.

Its CAML definition would be:

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

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

[1] CAML would not accept such a program since, because of pattern-matching, we cannot abstract over data constructors.

Part II

CAML Specifics

Chapter 7

Annotating type definitions

The definition of a sum or product type may be annotated in order to provide a special semantics to the type being defined. One may distinguish two (incompatible) annotations:

- `lazy` annotation which specifies a delayed evaluation of concerned expressions; it provides lazy evaluation and the ability of manipulating infinite data structures;
- `mutable` annotation allowing the (physically) destructive usage of data structures; they allow *imperative* programming in CAML.

We study in this chapter successively `lazy` and `mutable` annotations.

7.1 Lazy data structures

Lazy components of data structures are components which are evaluated if and only if their value is needed. This is the paradigm of lazy evaluation.

Suppose we want to define the infinite stream of natural numbers (for example). Such a definition would be impossible in CAML without a heavy encoding of the tails of streams into functional values.

If we try the naive way, we get:

```
#type 'a stream = {Hd:'a; Tl: 'a stream};;
Type stream defined
     .Hd : ('a stream -> 'a)
   ; .Tl : ('a stream -> 'a stream)
```

We define here a generic data structure which contains only infinite streams. That data structure fits our needs, except for the fact that we are unable to build an actual stream: the compiler is not strong enough to produce code for building such a structure. Moreover, the semantics of call-by-value implies a loop in building such infinite data structures.

It is thus better in this case to use *lazy evaluation*, in order to do as little work as possible. This is the role of the `lazy` annotation.

Let us work on a simpler example. We define the generic type of possibly delayed values (*frozen values*).

```
#type 'a frozen = lazy 'Frozen of 'a;;
Type frozen defined
     'Frozen : ('a -> 'a frozen)
```

We used the keyword `lazy` in order to give a lazy semantics to the `'Frozen` data constructor. When given an expression as argument, the `'Frozen` constructor will not evaluate it, it will instead inject that expression unevaluated.

```
#let x= 'Frozen (100*5);;
Value x = ('Frozen *) : num frozen

#x;;
('Frozen *) : num frozen
```

The CAML system prints a "`*`" because it does not know the value under `'Frozen`: that value has not been computed yet.

If now, we try to use the result of the delayed expression, we get:

```
#match x with 'Frozen n -> n+1;;
501 : num

#x;;
('Frozen 500) : num frozen
```

Now, the expression whose evaluation had been delayed has been evaluated once. After that evaluation, the value `x` has been updated with the new value, and it will not be computed again: its result has been made available by the first evaluation. Only the first access to delayed expressions implies evaluation: subsequent accesses will find a regular value.

Now, let us come back to the problem of building infinite streams. Here is the type we want to define:

```
#type 'a stream = {Hd:'a; lazy Tl: 'a stream};;
Warning: type stream redefined
Type stream defined
     .Hd : ('a stream -> 'a)
   ; .Tl : ('a stream -> 'a stream)
```

We choose to be lazy on tails of streams; we could have chosen to be lazy on both heads and tails. The current definition is sufficient for our needs.We need now a functional on streams analoguous to `map`:

```
#let map_stream f = map_rec
#where rec map_rec {Hd=x; Tl=L} =
#                  {Hd = f(x); Tl = map_rec(L)};;
Value map_stream = <fun> :
      (('a -> 'b) -> 'a stream -> 'b stream)

#let rec nth_stream s =
```

```
#    function 0 -> s.Hd
#           | n -> nth_stream s.Tl (n-1);;
Value nth_stream = <fun> : ('a stream -> num -> 'a)
```

We are now able to define our stream. We give a recursive definition, and we enclose the defined stream into the `'Frozen` constructor in order to make sure that the compiler will be able to work without risk:

```
#let rec ('Frozen nats) =
#         'Frozen {Hd=0;
#                  Tl = map_stream succ nats};;
Value nats = {Hd=0; Tl=*} : num stream

#nth_stream nats 4;;
4 : num

#nats;;
{Hd=0; Tl={Hd=1; Tl={Hd=2; Tl={Hd=3; Tl={Hd=4; Tl=*}}}}} :
 num stream
```

After evaluation of suspended expressions, updating occurs: this allows the computation of the stream of natural numbers in a progressive way, and the access without evaluation of values which have been already computed.

The Miranda family of functional languages uses lazy evaluation, and many interesting examples of lazy programs may be found in the literature.

7.2 An example of lazy data structures: formal series

We give in this section an example of the power of expression of lazy data structures. Formal series will be represented by infinite data structures (streams) composed of their coefficients. The index of a coefficient in a stream will represent its exponent.

```
#type series = {lazy Const : num; lazy Rest : series};;
Type series defined
     .Const : (series -> num)
   ; .Rest : (series -> series)
```

We now define the constant series:

```
#let rec constant n = {Const = n; Rest = constant n};;
Value constant = <fun> : (num -> series)
```

The constant series `zero` can be defined by:

```
#let zero = constant 0;;
Value zero = {Const=*; Rest=*} : series
```

The opposite of a series:

```
#let rec opposite {Const = c; Rest = t} =
#    {Const = -c; Rest = opposite t};;
Value opposite = <fun> : (series -> series)
```

The addition of two series:

```
#let rec add {Const = n; Rest = s1}
#            {Const = m; Rest = s2} =
#         {Const = n+m; Rest = add s1 s2};;
Value add = <fun> : (series -> series -> series)
```

Many more tools for manipulating formal series may be defined (see the exercices at the end of this chapter). We now concentrate on the definition of some well-known functions represented by their formal series. After the definition of the integration of formal series, we will define the formal series of sine and cosine.

In order to integrate a formal series, we need to specify the constant part of the result. The `integrate` function will return only the `Rest` part of the result: we will have to enclose it in a `{Const = c; Rest = ...}` construction in order to use `c` as the constant part.

```
#let integrate = integrate_at_index 1
#where rec integrate_at_index n {Const = c; Rest = t} =
#        {Const = c/n;
#         Rest = integrate_at_index (n+1) t};;
Value integrate = <fun> : (series -> series)
```

We can now define `sine` and `cosine` in a mutually recursive way:

```
#let rec ('Frozen ('Frozen sine, 'Frozen cosine)) =
#         'Frozen ('Frozen {Const = 0;
#                           Rest = integrate cosine},
#                  'Frozen {Const = 1;
#                           Rest = integrate
#                                  (opposite sine)});;
Value sine = {Const=0; Rest=*} : series
Value cosine = {Const=1; Rest=*} : series
```

We need a printing function for formal series. That function needs an index in order not to loop indefinitely:

```
#let rec print_series_until n {Const = c; Rest = s} =
#        if n <= 0 then print_newline()
#        else (print_num c; print_string " ";
#              print_series_until (n-1) s)
#;;
Value print_series_until = <fun> : (num -> series -> unit)

#print_series_until 6 zero;;
0 0 0 0 0 0
() : unit
```

Let us print the first five coefficients of `sine` and compare the evaluation degree of `cosine` before and after:

```
#cosine;;
{Const=1; Rest=*} : series

#print_series_until 6 sine;;
0 1 0 -1/6 0 1/120
() : unit

#cosine;;
{Const=1;
 Rest=
  {Const=0;
   Rest=
    {Const=-1/2;
     Rest=
      {Const=0; Rest={Const=1/24; Rest={Const=*; Rest=*}}}}}} :
 series
```

This shows the internal sharing between `sine` and `cosine`: when we ask for more coefficients to `sine`, `sine` needs some intermediate values to be computed by `cosine`.

We terminate here our example of formal series. A further development of this example is an interesting exercice showing the power of lazy evaluation.

7.3 Mutable data structures

In imperative languages such as Pascal, it is natural to write values into memory locations. In Lisp, the dangerous functions `rplaca` and `rplacd` which permit to physically modify lists are of great use when efficiency is to be taken into account.

Some programs would be hard to write if imperative programming style was forbidden. When one wants to program a state machine, then it is natural to represent a state as a mutable data structure, for example.

In the same way as for lazy data structures, it is possible in CAML to specify, in the definition of a data structure, that some parts of that data structure are mutable. This is useful only when defining product types, and we will limit ourselves to that case. A full description of mutable data structures may be found in the CAML Reference Manual.

7.3.1 User-defined mutable data structures

Suppose 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. Here is how we can do that:

```
#type person =
#    {Name: string; mutable Age: num;
```

```
#     mutable Job: string; mutable City: string};;
Type person defined
     .Name : (person -> string)
   ; .Age : (person -> num)
   ; .Job : (person -> string)
   ; .City : (person -> string)
```

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

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

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:

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

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

```
#jean.Age <- jean.Age + 1;;
24 : num
```

Now, the value `jean` has been modified into:

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

We may try to change the `Name` of `jean`, but we won't succeed: the typecheker will not allow us to do it:

```
#jean.Name <- "Paul";;

illegal use of an occurrence with the non mutable label
Name in jean.Name <- "Paul"
1 error in typechecking

Typecheck Failed
```

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

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

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 from writing:

```
#let get_older p =
#    match p with {Age=n; _} -> p.Age <- n + 1;;
Value get_older = <fun> : (person -> num)
```

Other examples would be:

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

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

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 only when in the presence of *side-effects* as physical modifications and input/output.

7.3.2 The `ref` type

The `ref` type is the predefined type of a mutable data structure. It is present in the CAML system for reasons of compatibility with earlier CAML versions.The `ref` type could be defined as (we don't use the `ref` name in the following definition because we want to preserve the original `ref` type):

```
#type 'a reference = {mutable Ref: 'a};;
Type reference defined
     .Ref : ('a reference -> 'a)
```

Example of building a value of type `ref`:

```
#let r = ref (1+2);;
Value r = (ref 3) : num ref
```

The definition of `r` should be read as "let `r` be a reference on the value of `1+2`". The value of `r` is nothing but a memory location inside of which we can write.

We consult a reference (i.e. read from its memory location) with the "`!`" symbol:

```
#!r + 1;;
4 : num
```

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

```
#r:=!r+1;;
4 : num

#r;;
(ref 4) : num ref
```

The `ref` identifier acts as a sum data constructor but uses a special syntax.

7.3.3 Polymorphism and mutable data structures

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

```
#let r = ref [];;

line 1: cannot generalize type 'a list
for argument of mutable sum constructor ref
1 error in typechecking

Typecheck Failed
```

The reason is that once the type of `r` has been computed (`('a list) ref`), 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).

7.4 Mutability and laziness

Mutable data structures allow imperative programming in CAML while lazy data structures opens the door to lazy evaluation. These two programming styles are incompatible, and the programmer is in charge to clearly separate lazy and side-effecting parts of his/her programs. Lazy evaluation and side-effects make the prediction of program evaluation very hard. However, each programming style has different applications, and mixing both styles requires a lot of imagination.

Exercises

7.1 Compute the stream of prime numbers by using Erathostene's sieve and lazy streams.

7.2 Define the derivative of formal series.

7.3 Define the multiplication function of formal series.

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

Chapter 8

Escaping from computations: exceptions

In some situations, it is necessary to escape from computations. If we are trying to compute the integer division of an integer `n` by `0`, then we must escape from that embarassing situation without returning any result.

Another example of the usage of such an escape mechanism appears when we want to define the `head` function on lists:

```
#let head = function
#     x::L -> x
#   | [] -> raise failure "head: empty list";;
Value head = <fun> : ('a list -> 'a)
```

We cannot give a regular value to the expression `head []` without losing the polymorphism of `head`. We thus choose to escape: we *raise an exception.*

8.1 Exceptions

An exception may be seen as a special kind of value, although they are not values in CAML (just wait for future versions of CAML). An exception:

- has a *name* (`failure` in our example),
- and holds values (`"head:  empty list"` of type `string` in the example).

An exception is:

- either predefined (like `failure`),
- or globally defined,
- or locally declared.

In order to locally declare a new exception, we must use the following construct:

```
exception <exc. name> [of <type>] in <expression>
```

The part "`of <type>`" is optional and "`of unit`" is the default.

Usage of local exceptions is not encouraged because exceptions are dynamic entities (they may escape from their scope, when this scope is local), and it may be hard to find the name of a local exception raised back to toplevel.

In the following, we will only consider global exceptions. A global exception is defined as a local one, but without the "`in <expression>`" part.

Example:

```
#exception Found of num;;
Exception Found of num defined
```

The exception `Found` has been declared, and it will hold values of type `num`.

8.2 Raising an exception

Raising an exception is done with the following construct:

```
raise <exception name> <expression>
```

The `raise` keyword will raise the exception whose name has been specified with the value of `<expression>`. The type of the considered expression must agree with the type of values held by the considered exception.

Example:

```
#let find_index p = find 1
#where rec find n =
#   function [] -> raise failure "not found"
#          | x::L -> if p(x) then raise Found n
#                    else find (n+1) L;;
Value find_index = <fun> : (('a -> bool) -> 'a list -> 'b)
```

The `find_index` function always fails. It raises:

- `Found n`, if there is an element `x` of the list such that `p(x)`, in this case `n` is the index of `x` in the list,
- the `failure` exception if no such `x` has been found.

Raising exceptions is more than an error mechanism: it is a programmable control structure. In the `find_index` example, there was no error when raising the `Found` exception: we only wanted to quickly escape from the computation, since we found what we were looking for. This is why it must be possible to *trap* exceptions: we want to trap possible errors, but we also want to get our result in the case of the `find_index function`.

8.3 Trapping exceptions

Trapping exceptions is realized with the following construct:

```
try <expression> with <match cases>
```

It evaluates `<expression>`, if no exception is raised during the evaluation, then the result of the `try` construct is the result of `<expression>`. If an exception is raised during this evaluation, then the raised exception is matched against the `<match cases>`. If a case matches, then control is passed to it, if no case matches, then the exception is propagated outside of the `try` construct, looking for the enclosing `try`.

Example:

```
#let find_index p L =
#   (try find 1 L with Found n -> n)
#   where rec find n =
#   function [] -> raise failure "not found"
#          | x::L -> if p(x) then raise Found n
#                    else find (n+1) L;;
Value find_index = <fun> : (('a -> bool) -> 'a list -> num)

#find_index (function n -> (n mod 2) = 0)
#           [1;3;5;7;9;10];;
6 : num
```

The `<match cases>` part of the `try` construct contains match cases on "exceptional values". It is thus possible to trap any exception by using the _ symbol. As an example, the following function traps any exception raised during the application of its two arguments. Warning: the _ will also trap interruptions from the keyboard such as `^C`!

```
#let catch_all f arg default =
#       try f(arg) with _ -> default;;
Value catch_all = <fun> : (('a -> 'b) -> 'a -> 'b -> 'b)
```

This function could be refined in order to have a special treatment of keyboard interruptions:

```
#let catch_all f arg default =
#    try f(arg)
#    with break _ -> message "catch_all interrupted"
#                    reraise
#       | _ -> default;;
Value catch_all = <fun> : (('a -> 'b) -> 'a -> 'b -> 'b)
```

The `reraise` keyword is used when one wants to trap an exception, do something special (close input/output channels, for example), and then raise again that exception. In the example above, we printed a message to the standard output channel (the terminal), before raising again the `break` exception.

```
#catch_all (function x -> raise break ()) 1 0;;
catch_all interrupted

Interrupted
```

8.4 Polymorphism and exceptions

Exceptions must not be polymorphic for a reason similar to the one for references (although it is a bit harder to give an example).

```
#exception Exc of 'a list;;

line 1: cannot generalize type 'a list
for exception Exc in exception Exc of 'a list
1 error in typechecking

Typecheck Failed
```

The reason is that if the exception `Exc` is declared to be polymorphic, then a function may raise `Exc [1;2]`. Then, another function may trap that exception, obtaining the value `[1;2]` whose real type is `num list`. But the only type known by the typechecker is `'a list`. It may then be possible to build an ill-typed CAML value `[true; 1; 2]`, since the typechecker does not possess any further type information than `'a list`.

Exercises

8.1 Define the function `find_succeed` which given a function `f` and a list `L` returns the first element of `L` on which the application of `f` succeeds.

8.2 Define the function `map_succeed` which given a function `f` and a list `L` returns the list of the results of successful applications of `f` to elements of `L`.

Chapter 9

Dynamic types

Typechecking in CAML occurs statically, at compile-time. However, Milner's type system is not powerful enough in order to give a type to an evaluation function. Such a function (let's call it `eval_syntax`) takes as argument a piece of CAML abstract syntax. `ML` is the type of these data structures (they are CAML values, since most of CAML is written in CAML).

But what should be the codomain of `eval_syntax`? In fact, the type of the result depends on the result itself, since typechecking is part of the evaluation. It could be `num` or `string` or any other type. Warning: this does not imply that we have:

```
eval_syntax : ML -> 'a
```

Given a CAML abstract syntax tree `t`, we do not have `(eval_syntax t : 'a)`, since `(eval_syntax t)` may be `num` *or* `string` or any other type, but *not all of them.* Since the codomain of `eval_syntax` is a union of types, it must be the *disjoint union of all (present and future) CAML types.*

Since CAML sum typesare finite disjoint unions, such a type is not definable in CAML It is thus predefined in CAML and called `dyn`.

We saw that CAML sum types were labelled. In the case of `dyn`, the labels that we use are the CAML types themselves. Since we have the CAML typechecker at our disposal, we simply use it to "label" values of type `dyn`.

In the following, we will call *dynamic values* or simply *dynamics* the values of type `dyn` (i.e. the values with dynamic types). We use the word "dynamic" because such a type leads to dynamic typechecking, as we will see later.

9.1 Creation of dynamics

Dynamics are created with the following CAML construct:

```
dynamic <expression>
```

The effect of the `dynamic` keyword is to ask the typechecker to enclose the type of `<expression>` and its value inside of a single value of type `dyn`. This is nothing but the injection of the value of `<expression>` into the type `dyn`, the injection being the type of `<expression>` itself.

```
#let d1 = dynamic (1+2);;
Value d1 = (dynamic (3 : num)) : dyn
```

CAML prints dynamics in a special way in order for the user to be able to read the internal value and type of the dynamic in the same way as any other value.

9.2 Dynamics and polymorphism

Here again, there is a restriction in the creation of dynamics. When doing so, the type of the internal value of the dynamic must be completely determined. This is why CAML accepts:

```
#let d2 = dynamic (function x -> x);;
Value d2 = (dynamic (<fun> : ('a -> 'a))) : dyn
```

and refuses:

```
#function x -> dynamic x;;

line 1: cannot generalize type 'a
for dynamic expression: dynamic x
1 error in typechecking

Typecheck Failed
```

In the second example, the type of `x` (i.e. the internal type of `(dynamic x)`) still depends on the type of the `x` formal parameter. The CAML typechecker cannot in this case compute the type of `x`, which will serve as a label for the dynamic value.

9.3 Extraction of dynamic values

Once again, pattern-matching is used to extract internal values and types of dynamic values.

When matching values belonging to sum types, run-time tests usually have to be done on the constructors of values (injections). Since the type `dyn` is a sum type, run-time tests have also to be done. But, since the data constructors are CAML types, pattern-matching of dynamic values is nothing but *dynamic typechecking*.

```
#match d1
#with dynamic(b:bool) -> if b then 1 else 2
#   | dynamic 0 -> 0
#   | dynamic(n:num) -> n+1
#   | d -> message "Unexpected dynamic value";
#          print d; print_newline(); -1;;
4 : num
```

The previous example should be clear: if the internal value of `d1` is any boolean value, then we test that value and return `1` or `2` following the result of the test. If the internal value is `0` (the fact that `(0 : num)` is deduced by the typechecker), then return `0`.

If the internal value is any other numeric value, then return its successor. Otherwise, we print a message, then we print the dynamic value argument, print a newline in order to terminate printing, and return the number `-1`.

So far, this can be seen as regular pattern-matching. But, consider the case of the dynamic value `d2`, whose internal value is the (polymorphic) identity function. When we match on a dynamic value `d`, against the pattern `dynamic (v:t)` where `t` is a CAML type, we must check that the internal value of `d` is of type `t`. In the case of `d2` the type of the internal value is `'a -> 'a`. If `t` is `num -> num`, then the internal value of `d2` *is also* of type `num -> num`. Then the pattern-matching must succeed in this case.

```
#match d2
#with dynamic (f: num -> bool)
#        -> if f(1) then 0 else 2
#   | dynamic (f : num -> num) -> f 10
#   | dynamic (f : 'a -> 'a) -> f 1000;;
Warning: 1 partial match in this phrase
10 : num
```

This is a feature which is not easy to use, but in most cases, we know exactly the internal type of the dynamic values we manipulate, and we practically never enter such situations.

Further information about values with dynamic types can be found in [Weis *et al.*, 1990] and [Leroy and Mauny, 1991].

Chapter 10

Grammars

CAML possesses an interface with the Yacc ([Johnson, 1986]) parser generator. One may thus associate a concrete syntax to any CAML data type. Such an association is called an *object-language* (it is in fact only the concrete aspect of such a language).

The programmer must specify how to recognize such a concrete syntax, and what has to be done when an object-language construct has been recognized. A grammar definition consists thus of:

- a *grammatical* part, specifying what phrases belong to the object language,
- a *semantic* part, specifying what actions have to be realized when a part of a phrase has been recognized.

A grammar definition is a CAML toplevel construct (there is no local grammar).

The grammatical part of a grammar "does something like" pattern-matching. A parser (a value produced by a grammar definition) matches the input stream in order to recognize some "patterns".

When defining a grammar, the user has access to "predefined patterns" matching numbers, string constants, boolean values and CAML identifiers. Let us see an example: a grammar for a type `person`.

```
#type person =
#     {Name:string; Age: num; Job:string};;
Type person defined
     .Name : (person -> string)
   ; .Age : (person -> num)
   ; .Job : (person -> string)

#grammar for values P =
#rule entry p =
#  parse IDENT(s); Literal "is"; NUM(n);
#        Literal "and"; Literal "is"; IDENT(j)
#           -> {Name=s; Age=n; Job=j};;
Calling Yacc ... ......................................
Value P = <fun> : (string -> Parsers)
Grammar P for values defined
```

```
entry p : person
```

In the grammar `P`, there is only one entry point `p`. An *entry point* corresponds to a parsing function. A grammar may contain several entry points, i.e. several parsers defined in a mutually recursive way.

When called on an input stream, the parser `p` of `P` will try to:

- recognize an identifier (`IDENT`), and name `s` the *semantic value* of that identifier (its name of type `string`) – if no identifier is found, then a *parsing error* is raised (this corresponds to a match failure);
- then, the parser will check that "`is`" follows the identifier ("`is`" is a keyword of our object language) – if no "`is`" is found, then a parsing error is raised;
- then the parser should find a number, naming `n` its semantic value (of type `num`;
- then the parser checks that "`and`" and "`is`" follow the number;
- finally, an identifier must be found, the parser will name `j` its semantic value.
- Now, the "pattern-matching" (parsing) has succeeded, the result of the parsing will be the right part of the rule (i.e. the CAML expression that follows the "`->`"), in this case, an object of type `person`.

When CAML evaluates a grammar definition, it does two things:

- it produces a new value, `P` in our example (we will see later how to use this value). That value may be used in order to access the parsing functions of that grammar.
- it enriches the set of known object-languages by a new language: `P`, with `p` as single entry point.

But how do we ask CAML to parse an object language `L` at entry point `E`? We use *angle brackets* (i.e. "`<<`" and "`>>`") in order to surround object-languages phrases. And we use the first angle bracket in a special way: we write "`<:L:E<`" in order to specify that what follows will be a phrase of language `L` at entry point `E`. In our example, the language is `P` and the entry point is `p`:

```
#<:P:p< Jean is 21 and is student >>;;
{Name="Jean"; Age=21; Job="student"} : person
```

CAML also possesses a notion of *default object language*. It is usually the last language whose grammar has been defined (`P:p` in our case).In order to write a phrase belonging to the default language, we use regular angle brackets:

```
#<< Paul is 67 and is retired >>;;
{Name="Paul"; Age=67; Job="retired"} : person
```

Sometimes, of course, parsing may fail:

```
#<<Jean is 25 and is a teacher>>;;

line 1 Syntax error:
Skipping: teacher >> ;;
Parse Failed
```

Here, our grammar was not sophisticated enough to take articles into account. Let us complicate it a little bit:

```
#grammar for values P =
#rule entry p =
#  parse IDENT(s); Literal "is"; NUM(n);
#        Literal "and"; Literal "is"; article _; IDENT(j)
#           -> {Name=s; Age=n; Job=j}
#and article =
#  parse (* nothing *) -> ()
#      | Literal "a" -> ()
#      | Literal "an" -> ()
#      | Literal "one" -> ()
#;;
Calling Yacc ... ........................................
Value P = <fun> : (string -> Parsers)
Grammar P for values defined
  entry p : person
```

The *non-terminal* (as in parsing terminology) `article` checks the presence of an article in the phrase, but does not return any interesting result. Furthermore, it is not specified to be an entry point: we don't have access to its parser by "`<:P<article<`".

Our language has been enriched by the presence of an optional article before the job of the person:

```
#[ <<Jean is 23 and is student>>;
#  <<Paul is 27 and is a teacher>> ];;
[{Name="Jean"; Age=23; Job="student"};
 {Name="Paul"; Age=27; Job="teacher"}] : person list
```

Our last example was very simple, and we did not gain much time in writing objects of type `person` with their `P:p` concrete sytax, instead of writing their regular CAML form. The next example will be more interesting in that it concerns our simple language for arithmetic expressions (enriched with a factorial expression), and it will be much faster to write `<<1+2>>` instead of `'Plus {Arg1='Const 1; Arg2='Const 2}`. For that example, we will also discuss problems of operator precedences and associativity.

10.1 Using Yacc features

Since the grammar construct is an interface to the Yacc parser generator, we will use the Yacc facilities of precedence and associativity specification. When declaring a grammar, one may use

some of the Yacc features in order to specify the relative precedences and associativity rules for operators. We first define the type of arithmetic expressions, and then we give them a (classical) concrete syntax by a grammar definition.

```
#type arith_expr = 'Const of num
#                | 'Plus of args
#                | 'Mult of args
#                | 'Minus of args
#                | 'Div of args
#                | 'Fact of arith_expr
#and args = {Arg1:arith_expr; Arg2:arith_expr};;
Type arith_expr defined
     'Const : (num -> arith_expr)
   | 'Plus : (args -> arith_expr)
   | 'Mult : (args -> arith_expr)
   | 'Minus : (args -> arith_expr)
   | 'Div : (args -> arith_expr)
   | 'Fact : (arith_expr -> arith_expr)
Type args defined
     .Arg1 : (args -> arith_expr)
   ; .Arg2 : (args -> arith_expr)

#grammar for values AExpr =
#precedences
#   left "+" "-"; (* + and - have same precedence *)
#                 (* and associate to the left    *)
#   left "*" "/"; (* Idem for * and / *)
#
#rule entry Expr =
#  parse Expr(e1); "-"; Expr(e2)
#                        -> 'Minus {Arg1=e1; Arg2=e2}
#      | Expr(e1); "+"; Expr(e2)
#                       -> 'Plus {Arg1=e1; Arg2=e2}
#      | Expr(e1); "*"; Expr(e2)
#                       -> 'Mult {Arg1=e1; Arg2=e2}
#      | Expr(e1); "/"; Expr(e2)
#                        -> 'Div {Arg1=e1; Arg2=e2}
#      | Expr0(e) -> e
#
#and entry Expr0 =
#  parse NUM (n) -> 'Const n
#      |  "!"; Expr0(e) -> 'Fact e
#      | "("; Expr(e); ")" -> e
#      | "end" -> raise break () ;;
Calling Yacc ... ..............................................
Value AExpr = <fun> : (string -> Parsers)
```

```
Grammar AExpr for values defined
  entry Expr : arith_expr
  entry Expr0 : arith_expr
```

The keywords `precedences`, `left` and `right` specify the associativity rules of operators. Their order of appearance specify their relative precedences.

We may now write arbitrarily complex expressions using our concrete syntax:

```
#<<1+2*3-!4>>;;
('Minus
 {Arg1=
   ('Plus
    {Arg1=('Const 1);
     Arg2=('Mult {Arg1=('Const 2); Arg2=('Const 3)})});
  Arg2=('Fact ('Const 4))}) :
 arith_expr

#<<(1+2)*(3-!4)>>;;
('Mult
 {Arg1=('Plus {Arg1=('Const 1); Arg2=('Const 2)});
  Arg2=('Minus {Arg1=('Const 3); Arg2=('Fact ('Const 4))})}) :
 arith_expr
```

For a complete description of Yacc features available in CAML, the reader is invited to consult the CAML documentation and the Yacc manual.

10.2 Using produced parsers

The value (of type `string -> Parsers`) defined by a grammar definition is a functional value usable in order to give access to *parsing functions*. They work as association lists: they associate a set of parsers to the name of each entry point:

```
#AExpr "foo";;

Evaluation Failed: AExpr does not have entry point named "foo"

#AExpr "Expr";;
{Parse=<fun>; Parse_string=<fun>; Parse_channel=<fun>;
 Parse_raw=<fun>} : Parsers
```

The type `Parsers` is a predefined CAML (product) type. A value of type `Parsers` is a record containing parsing functions which take their input from different input channels. We will use only the interactive parser (the one taking its input on the standard input channel). For more information about the other parsing functions, see the CAML documentation.

The rest of this section is dedicated to a simple interpreter of arithmetic expressions.

We first define a semantic function for arithmetic expressions. But, first of all, the factorial function:

```
#let rec fact n = if n <= 0  then 1
#                 else n * fact(n-1);;
Value fact = <fun> : (num -> num)
```

Then, we give the semantic function:

```
#let rec sem = function
#    'Const n -> n
#  | 'Plus {Arg1=e1;Arg2=e2} -> sem(e1) + sem(e2)
#  | 'Mult {Arg1=e1;Arg2=e2} -> sem(e1) * sem(e2)
#  | 'Minus{Arg1=e1;Arg2=e2} -> sem(e1) - sem(e2)
#  | 'Div  {Arg1=e1;Arg2=e2} -> sem(e1) / sem(e2)
#  | 'Fact(e) -> fact (sem e);;
Value sem = <fun> : (arith_expr -> num)

#sem <<1+2*3-!(1+2)>>;;
1 : num
```

Now, we build a toplevel loop for our small language:

```
#let arith_parser =
#    let p = (AExpr "Expr0").Parse
# in function () ->
#             (match p ()
#              with MLquote(dynamic(e:arith_expr)) -> e
#                 | _ -> raise failure "Bug!")
#;;
Value arith_parser = <fun> : (unit -> arith_expr)

#let rec parse_and_eval =
#    let eval = sem o arith_parser
#    in function () ->
#           let result = eval ()
#           in print_string "=> ";
#              print_num result;
#              print_newline();
#              parse_and_eval ();;
Value parse_and_eval = <fun> : (unit -> 'a)
```

A much more sophisticated toplevel loop could be constructed. See the CAML Reference manual in order to find examples of "clean" user-defined toplevel loops.

Let us try a session of our arithmetic calculator:

```
#parse_and_eval();;
#(1+2)
=> 3
#!(5+5)
=> 3628800
```

```
#(2*3*4*5*6*7)
=> 5040
#(1+2*1+2)
=> 5
#end

Interrupted
```

10.3 Other features and pitfalls

There are lots of pitfalls in the Yacc interface which are due to the current implementation of grammars (mainly because of the usage of Yacc). Users who want to build real parsers (and this is possible, since the CAML parser itself is a CAML grammar), are invited to read the CAML documentation. Some patience is needed in order to be able to use all the features and misfeatures of the CAML-Yacc interface.

The interested reader is invited to consult the CAML documentation ([Weis *et al.*, 1990])and the Yacc documentation ([Johnson, 1986]). A better integration of parsers into functional languages is suggested in [Mauny, 1989].

Part III

A Complete Example

Chapter 11

ASL: A Small Language

We present in this chapter a simple language: ASL (A Small Language). This language is in fact a λ-calculus, enriched with a conditional construct. We need such a conditional, because our language will be submitted to call-by-value: thus, the conditional cannot be a function.

ASL programs are built up from numbers, variables, functional expressions (λ-abstractions), applications and conditionals. An ASL program consists of a global declaration of an identifier getting bound to the value of an expression. The primitive functions which are available are equality between numbers and arithmetic binary operations.

We start by defining the abstract syntax of ASL expressions and of ASL toplevel phrases. Then we define a grammar in order to give a concrete syntax to ASL programs.

11.1 ASL abstract syntax trees

We encode variable names by numbers. These numbers represent the *binding depth* of variables. For instance, the ASL function of `x` returning `x` (the ASL identity function) will be represented as:

```
'Abs("x", 'Var 1)
```

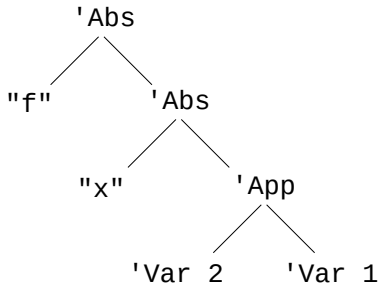
And the ASL application function which would be written in CAML:

```
(function f -> (function x -> f(x)))
```

would be represented as:

```
'Abs("f", 'Abs("x", 'App('Var 2, 'Var 1)))
```

and should be viewed as the tree:



`'Var n` should be read as "an occurrence of the variable bound by the nth abstraction node encountered when going toward the root of the abstract syntax tree". In our example, when going from `'Var 2` to the root, the 2nd abstraction node we encounter introduces the `"f"` variable.

The numbers encoding variables in abstract syntax trees of functional expressions are called "De Bruijn[1] numbers". The strings that we attach to abstraction nodes simply serve as documentation: they will not be used by any of the semantic analyses that we will perform on the trees. The type of ASL abstract syntax trees is defined by:

```
#type asl = 'Const of num
#         | 'Var of num
#         | 'Cond of asl * asl * asl
#         | 'App of asl * asl
#         | 'Abs of string * asl
#
#and top_asl = 'Decl of string * asl;;
Type asl defined
     'Const : (num -> asl)
   | 'Var : (num -> asl)
   | 'Cond : (asl * asl * asl -> asl)
   | 'App : (asl * asl -> asl)
   | 'Abs : (string * asl -> asl)
Type top_asl defined
     'Decl : (string * asl -> top_asl)
```

11.2 Parsing ASL programs

Now we come to the problem of defining a concrete syntax for ASL programs and declarations.

The choice of the concrete aspect of the programs is simply a matter of taste. The one we choose here is close to the syntax of λ-calculus (except that we will use the *backslash* character because there is no "λ" on our keyboards). We will use the *curried* versions of equality and arithmetic functions. We will also use a *prefix* notation (à la Lisp) for their application. We will write "`+ (+ 1 2) 3`" instead of "`(1+2)+3`".

The final output of our grammar rules will be abstract syntax trees of type `asl` or `top_asl`. This implies that we will detect unbound identifiers at parse-time. In this case, we will raise the `Unbound` exception defined as:

```
#exception Unbound of string;;
Exception Unbound of string defined
```

We also need a function which will compute the binding depths of variables. That function simply looks for the position of the first occurrence of a variable name in a list. It will raise `Unbound` if there is no such occurrence. The `index` function is a predefined CAML function.

```
#let binding_depth s rho =
```

[1] They have been proposed by N.G. De Bruijn in [De Bruijn, 1962] in order to facilitate the mechanical treatment of λ-calculus terms.

```
#        try 'Var(index s rho)
#        with failure _ -> raise Unbound s;;
Value binding_depth = <fun> :
      (string -> string list -> asl)
```

We also need a global environment, containing names of already bound identifiers. The global environment contains predefined names of the equality and arithmetic functions. We represent it as a reference since each ASL declaration will augment it with a new name.

```
#let init_env =  ["+";"-";"*";"/";"="];;
Value init_env = ["+"; "-"; "*"; "/"; "="] : string list

#let global_env = ref init_env;;
Value global_env = (ref ["+"; "-"; "*"; "/"; "="]) :
      string list ref
```

We now give the grammar of ASL programs:

```
#grammar for values asl =
#rule entry top =
#    parse "let"; IDENT(s); "="; expr(e); ";"  -> 'Decl(s,e)
#        | expr(e); ";" -> 'Decl("it",e)
#and entry expr =
#    parse Expr(e) -> (try e !global_env
#                      with Unbound s
#                       -> message ("Unbound ASL identifier: "^s);
#                          raise failure "ASL parsing")
#and Expr =
#    parse "\\"; IDENT(s); "."; Expr(e)
#                    -> (function rho -> 'Abs(s, e(s::rho)))
#        | Expr1 (e) -> e
#and Expr1 =
#    parse Expr1(e1); Expr0(e2)
#                  -> (function rho -> 'App(e1 rho, e2 rho))
#        | Expr0(e) -> e
#and Expr0 =
#    parse NUM(n) -> (function _ -> 'Const n)
#        | IDENT(s) -> binding_depth s
#        | "+" -> binding_depth "+" | "*" -> binding_depth "*"
#        | "/" -> binding_depth "/" | "-" -> binding_depth "-"
#        | "=" -> binding_depth "="
#        | "if"; Expr(e1); "then"; Expr(e2); "else"; Expr(e3); "fi"
#             -> (function rho -> 'Cond(e1 rho, e2 rho, e3 rho))
#        | "("; Expr(e); ")" -> e;;
Calling Yacc ... ....................................................
Value asl = <fun> : (string -> Parsers)
Grammar asl for values defined
```

```
  entry top : top_asl
  entry expr : asl
```

We may now try our grammar (we do not augment the global environment at each declaration: this will be realised after the semantic treatment of ASL programs).

```
#<<let f = \x.x;>>;;
('Decl ("f",('Abs ("x",('Var 1))))) : top_asl

#<<let x = + 1 ((\x.x) 2);>>;;
('Decl
 ("x",
  ('App
   (('App (('Var 1),('Const 1))),
    ('App (('Abs ("x",('Var 1))),('Const 2)))))) :
 top_asl

#<<let y = g 3;>>;;
Unbound ASL identifier: g
(while parsing grammar asl)

Evaluation Failed: ASL parsing

#<<let x = if 0 then + else - fi;>>;;
('Decl ("x",('Cond (('Const 0),('Var 1),('Var 2))))) :
 top_asl
```

Chapter 12

Untyped semantics of ASL programs

In this section, we give a semantic treatment of ASL programs. We will use *dynamic typechecking*, i.e. we will test the type correctness of programs during their interpretation.

12.1 Semantic values

We need a type for ASL semantic values (representing results of computations). A semantic value will be either a numeric value, or a CAML functional value from ASL values to ASL values.

```
#type semval = 'Numval of num
#            | 'Funval of semval -> semval;;
Type semval defined
     'Numval : (num -> semval)
   | 'Funval : ((semval -> semval) -> semval)
```

We now define two exceptions. The first one will be used when we encounter an ill-typed program and will represent run-time type errors. The other one is helpful in debugging: it will be raised when our interpreter (semantic function) goes into an illegal situation.

The two following exceptions will be raised in case of run-time ASL type error, and in case of bug of our semantic treatment:]]

```
#exception Illtyped;;
Exception Illtyped defined

#exception SemantBug of string;;
Exception SemantBug of string defined
```

We must give a semantic value to our basic functions (equality and arithmetic operations). The next function transforms a CAML function into an ASL value.

```
#let init_semantics caml_fun =
#    'Funval
#      (function 'Numval n ->
#         'Funval(function 'Numval m -> 'Numval(caml_fun(n,m))
#                        | _ -> raise Illtyped)
```

```
#              | _ -> raise Illtyped);;
Value init_semantics = <fun> :
      ((num * num -> num) -> semval)
```

Associate a caml function to each ASL predefined function:

```
#let caml_function = function
#    "+" -> prefix +
#  | "-" -> prefix -
#  | "*" -> prefix *
#  | "/" -> prefix /
#  | "=" -> (function (n,m) -> if n=m then 1 else 0)
#  | s -> raise SemantBug ("Unknown primitive: " ^ s);;
Value caml_function = <fun> : (string -> num * num -> num)
```

The global semantic environment will be a reference on the list of predefined ASL values.

```
#let init_sem =  map (init_semantics o caml_function)
#                    init_env;;
Value init_sem =
      [('Funval <fun>); ('Funval <fun>); ('Funval <fun>);
       ('Funval <fun>); ('Funval <fun>)] :
      semval list

#let global_sem = ref init_sem;;
Value global_sem =
      (ref
       [('Funval <fun>); ('Funval <fun>); ('Funval <fun>);
        ('Funval <fun>); ('Funval <fun>)]) :
      semval list ref
```

12.2 Semantic functions

Here is the semantic function: it computes the value of an ASL expression from an environment `rho`. The environment will contain the values of globally defined ASL values or of temporary ASL values. It is organized as a list, and the numbers representing variable occurrences will be used as indices into the environment.

```
#let rec semant rho = sem
#    where rec sem = function
#      'Const n -> 'Numval n
#    | 'Var(n) -> nth rho n
#    |' Cond(e1,e2,e3) ->
#       (match sem e1 with 'Numval 0 -> sem e3
#                        | 'Numval n -> sem e2
#                        | _ -> raise Illtyped)
#    | 'Abs(_,e') -> 'Funval(fun x -> semant (x::rho) e')
```

```
#    | 'App(e1,e2) -> (match sem e1
#                       with 'Funval(f) -> f (sem e2)
#                          | _ -> raise Illtyped)
#;;
Value semant = <fun> : (semval list -> asl -> semval)
```

The main function must be able to treat an ASL declaration, evaluate it, and update the global environments (`global_env` and `global_sem`).

```
#let semantics = function 'Decl(s,e) ->
#    let result = semant !global_sem e
#    in global_env:=s::!global_env;
#       global_sem:=result::!global_sem;
#       print_string ("ASL Value of "^ s ^ " is ");
#       print (dynamic result);
#       print_newline();;
Value semantics = <fun> : (top_asl -> unit)
```

12.3 Examples

```
#semantics <<let f=\x.+ x 1;>>;;
ASL Value of f is ('Funval <fun>)
() : unit

#semantics <<let I = \x.x;>>;;
ASL Value of I is ('Funval <fun>)
() : unit

#semantics <<let x = I (f 2);>>;;
ASL Value of x is ('Numval 3)
() : unit

#semantics <<(if x then (\x.x) else 2 fi) 0;>>;;
ASL Value of it is ('Numval 0)
() : unit
```

Chapter 13

Encoding of recursion

13.1 Fixpoint combinators

We saw that we did not have recursion in ASL. However, it is possible to encode recursion by defining a *fixpoint combinator*. A fixpoint combinator is a function F such that:

$$F\ M \text{ is equivalent to } M\ (F\ M) \text{ modulo evaluation rules.}$$

for any expression M.

When using call-by-value, any application of a fixpoint combinator F such that:

$$F\ M \text{ evaluates to } M\ (F\ M)$$

leads to non-termination of the evaluation (because evaluation of $(F\ \ M)$ leads to evaluating $(M\ (F\ M))$, and thus $(F\ M)$ again).

We will use the Z fixpoint combinator defined by:

$$Z = \lambda f.((\lambda x.\ f\ (\lambda z.\ (x\ x)\ z))(\lambda x.\ f\ (\lambda z.\ (x\ x)\ z)))$$

The fixpoint combinator Z has the particularity of being usable under call-by-value evaluation regime (in order to check that fact, it is necessary to know the evaluation rules of λ-calculus). More precisely, we have:

$$Z\ M \text{ evaluates to } M\ (\lambda z.\ (\lambda x.M\ (\lambda z.(x\ x)\ z))\ (\lambda x.M\ (\lambda z.(x\ x)\ z))\ z)$$

which is equivalent to $(M\ (\lambda z.(Z\ M)\ z))$. With the Z combinator, the λz abstraction protects the expression from looping when submitted to call-by-value semantics.

```
#semantics <<let Z = \f.((\x.f(\z.(x x)z))
#                        (\x.f(\z.(x x)z)));>>;;
ASL Value of Z is ('Funval <fun>)
() : unit
```

We are now able to define the ASL factorial function:

```
#semantics
#  <<let fact =
#          Z (\f.(\n. if = n 0 then 1
#                     else * n (f (- n 1)) fi));>>;;
ASL Value of fact is ('Funval <fun>)
() : unit
```

and the ASL Fibonacci function:

```
#semantics
# <<let fib =
#      Z (\f.(\n. if = n 1 then 1
#                 else if = n 2 then 1
#                      else + (f (- n 1)) (f (- n 2)) fi
#                 fi));>>;;
ASL Value of fib is ('Funval <fun>)
() : unit

#semantics <<fact 25;>>;;
ASL Value of it is ('Numval (155112100433309859840000000))
() : unit

#semantics <<fib 10;>>;;
ASL Value of it is ('Numval 55)
() : unit
```

13.2 Recursion as a primitive construct

Of course, in a more realistic prototype, we would extend the concrete and abstract syntaxes of ASL in order to support recursion as a primitive construct. We do not do it here because we want to keep ASL simple. This is an interesting non trivial exercise!

Chapter 14

Static typing, polymorphism and type synthesis

We now want to realise static typechecking of ASL programs, i.e. realise typechecking *before* evaluation, preventing the need for run-time type tests during evaluation of ASL programs.

Furthermore, we want to have *polymorphism* (i.e. allow the identity function, for example, to be applicable to any kind of data).

Type synthesis may be seen as a game. When learning a game, we must:

- learn the rules (what is allowed, and what is forbidden);
- learn a winning strategy.

In type synthesis, the rules of the game are called a *type system*, and the winning strategy is the typechecking algorithm.

In the following sections, we give the ASL type system, the algorithm and an implementation of that algorithm. Most of this presentation is borrowed from [Clément *et al.*, 1986].

14.1 The type system

The type system we will consider for ASL has been first given by Milner for a subset of the ML language (in fact, a superset of λ-calculus).

A *type* is:

- the type Number;
- or a type variable (α, β, ...);
- or $\tau_1 \rightarrow \tau_2$, where τ_i are types.

In a type, a type variable is an *unknown*, i.e. a type that we are computing. We will use τ, τ', τ_1, ..., as *metavariables*[1] representing types.

[1] A metavariables should not be confused with a *variable* or a *type variable*.

Example $(\alpha \to \text{Number}) \to \beta \to \beta$ is a type.
□

A *type scheme*, is a type where some variables are distinguished as being *generic*. We may represent type schemes as being of the following form:

$$\forall \alpha_1, \ldots, \alpha_n.\tau \text{ where } \tau \text{ is a type.}$$

Example $\forall \alpha.(\alpha \to \text{Number}) \to \beta \to \beta$ and $(\alpha \to \text{Number}) \to \beta \to \beta$ are type schemes.
□

We will use σ, σ', σ_1, ..., as metavariables representing type schemes. We may also write type schemes as $\forall \vec{\alpha}.\tau$. In this case, $\vec{\alpha}$ represent a (possibly empty) set of generic type variables. When the set of generic variables is empty, we write $\forall.\tau$ or simply τ.

We will write $FV(\sigma)$ for the set of *unknowns* occurring in the type scheme σ. Unknowns are also called *free variables* (they are not bound by a $\forall$ quantifier).

We also write $BV(\sigma)$ (*bound type variables of* σ) for the set of type variables occurring in σ which are not free (i.e. the set of variables universally quantified). Bound type variables are also said to be *generic*.

Example If σ denotes the type scheme $\forall \alpha.(\alpha \to \text{Number}) \to \beta \to \beta$, then we have:

$$FV(\sigma) = \{\beta\}$$

and

$$BV(\sigma) = \{\alpha\}$$

□

A *substitution instance* σ' of a type scheme σ is the type scheme $S(\sigma)$ where S is a substitution of types to *free* type variables appearing in σ. When applying a substitution to a type scheme, a renaming of some bound type variables may become necessary, in order to avoid the capture of a free type variable by a quantifier.

Example

- If σ denotes $\forall \beta.(\beta \to \alpha) \to \alpha$ and σ' is $\forall \beta.(\beta \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\gamma \to \gamma)\}$ (i.e. S substitutes the type $\gamma \to \gamma$ to the variable α..
- If σ denotes $\forall \beta.(\beta \to \alpha) \to \alpha$ and σ' is $\forall \delta.(\delta \to (\beta \to \beta)) \to (\beta \to \beta)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\beta \to \beta)\}$. In this case, the renaming of β into δ was necessary: we did not want the variable β introduced by S to be captured by the universal quantification $\forall \beta$.

□

The type scheme $\sigma' = \forall \beta_1 \ldots \beta_m.\tau'$ is said to be a *generic instance* of $\sigma = \forall \alpha_1 \ldots \alpha_n.\tau$ if there exists a substitution S such that:

- the domain of S is included in $\{\alpha_1, \ldots, \alpha_n\}$;

- $\tau' = S(\tau)$;
- no β_i occur free in σ.

In other words, a generic instance of a type scheme is obtained by giving more precise values to some generic variables, and (possibly) quantifying some of the new type variables introduced.

Example If $\sigma = \forall\beta.(\beta \to \alpha) \to \alpha$, then $\sigma' = \forall\gamma.((\gamma \to \gamma) \to \alpha) \to \alpha$ is a generic instance of σ. We changed β into $(\gamma \to \gamma)$, and we universally quantified on the newly introduced type variable γ.
□

We express this type system with the help of *inference rules*. An inference rule is written as a fraction:

- the numerator is called the *premisses*;
- the denominator is called the *conclusion*.

An inference rule:

$$\frac{P_1 \ \ldots \ P_n}{C}$$

may be read in two ways:

- "**If** $P_1, \ldots$ **and** P_n, **then** C".
- "**In order to prove** C, **it is sufficient to prove** $P_1, \ldots$ **and** P_n".

An inference rule may have no premisse: such a rule will be called an *axiom*. A complete proof will be represented by a *proof tree* of the following form:

$$\frac{\dfrac{\underline{P_1^m} \ \ldots \qquad \ldots \ \underline{P_l^k}}{\vdots}}{\dfrac{\underline{\quad} \ \ldots \ \underline{\quad}}{\dfrac{P_1^1 \ \ldots \ P_n^1}{C}}}$$

where the leaves of the tree ($P_1^m, \ldots, P_l^k$) are instances of axioms.

In the premisses and the conclusions appear *judgements* having the form:

$$\Gamma \vdash e : \sigma$$

Such a judgement should be read as "under the typing environment Γ, the expression e has type scheme σ". Typing environments are sets of *type hypotheses* of the form $x : \sigma$ where x is an identifier name and σ is a type scheme: typing environments give types to the variables occurring free (i.e. unbound) in the expression.

When typing λ-calculus terms, the typing environment is managed as a *stack* (because identifiers possess local scopes). We represent that fact in the presentation of the type system by *removing* the type hypothesis concerning an identifier name x (if such a type hypothesis exists) before adding a new type hypothesis concerning x.

We write $\Gamma - \Gamma(x)$ for the set ot type hypotheses obtained from Γ by removing the type hypothesis concerning x (if it exists).

Any numeric constant is of type Number:

$$\frac{}{\Gamma \vdash \texttt{'Const}\ n : \text{Number}}(\text{NUM})$$

We obtain type schemes for variables from the typing environment Γ:

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash \texttt{'Var}\ x : \sigma}(\text{TAUT})$$

It is possible to instanciate type schemes. The "GenInstance" relation represents generic instanciation.

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma' = \text{GenInstance}(\sigma)}{\Gamma \vdash e : \sigma'}(\text{INST})$$

It is possible to generalise type schemes with respect to variables which do not occur free in the set of hypotheses:

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma}(\text{GEN})$$

Typing a conditional:

$$\frac{\Gamma \vdash e_1 : \text{Number} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1 \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}) : \tau}(\text{IF})$$

Typing an application:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}(\text{APP})$$

Typing an abstraction:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

A special rule may be given in order to obtain polymorphism: this is the ML `let` construct.

$$\frac{\Gamma \vdash e : \sigma \qquad (\Gamma - \Gamma(x)) \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash (\lambda x\ .e')\ e : \tau}(\text{LET})$$

This type system has been proven to be *semantically sound*, i.e. the semantic value of a well-typed expression (an expression admitting a type) cannot be an *error value* due to a type error. This is usually expressed as:

> Well-typed programs cannot go wrong.

This fact implies that a clever compiler may produce code without any dynamic type test for a well-typed expression.

Example Let us check, using the above set of rules, that the following is true:

$$\emptyset \vdash \texttt{let f=}\lambda\texttt{x.x in f f}:\beta \to \beta$$

In order to do so, we will use the equivalence between the `let` construct and an application of an immediate abstraction (i.e. an expression having the following shape: $(\lambda v.M)N$. The (LET) rule will be crucial: without it, we could not check the above judgement.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x:\alpha}\text{TAUT}}{\emptyset \vdash (\lambda x.x):\alpha \to \alpha}\text{ABS}}{\emptyset \vdash (\lambda x.x):\forall\alpha.\alpha \to \alpha}\text{GEN} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash f:\forall\alpha.\alpha \to \alpha}\text{TAUT}}{\Gamma \vdash f:(\beta \to \beta) \to (\beta \to \beta)}\text{INST} \quad \dfrac{\dfrac{}{\Gamma \vdash f:\forall\alpha.\alpha \to \alpha}\text{TAUT}}{\Gamma \vdash f:\beta \to \beta}\text{INST}}{\Gamma = \{f:\forall\alpha.\alpha \to \alpha\} \vdash f\ f:\beta \to \beta}\text{APP}}{\emptyset \vdash (\lambda f.f\,f)(\lambda x.x):\beta \to \beta}\text{LET}$$

□

This type system does not tell us how to find the best type for an expression. But what is the best type for an expression? It must be such that any other possible type for that expression must be more specific; in other words, this type must be the *most general.*

14.2 The algorithm

How do we find the most general type for an expression of our language? The problem with the set of rules above, is that we could instanciate and generalise types at any time, introducing type schemes, while the most important rules (application and abstraction) used only types.

Let us write a new set of inference rules that we will read as an algorithm (close to a Prolog program):
Any numeric constant is of type Number:

$$\dfrac{}{\Gamma \vdash \texttt{'Const}\ n : \text{Number}}(\text{NUM})$$

The types of identifiers are obtained by taking generic instances of type schemes appearing in the typing environment. These generic instances wil be *types* and not type schemes. As it is presented (belonging to a deduction system), the following rule will have to anticipate the effect of the equality constraints between types in the other rules (multiple occurences of a type metavariable).

$$\dfrac{\tau = \text{GenInstance}(\sigma)}{\Gamma \cup \{x:\sigma\} \vdash \texttt{'Var}\ x:\tau}(\text{INST})$$

When we read this set of inference rules as an algorithm, the (INST) rule will be implemented by:

1. take as τ the type "most general generic instance" of σ,
2. make τ more specific by *unification* ([Robinson, 1971], [Huet, 1986]) when encountering equality constraints.

Typing a conditional:

$$\dfrac{\Gamma \vdash e_1 : \text{Number} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1 \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}) : \tau}(\text{COND})$$

Typing an application:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}(\text{APP})$$

Typing an abstraction:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \forall.\tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

Typing a `let` construct involves a generalisation step: we generalise as much as possible.

$$\frac{\begin{array}{ll}\Gamma \vdash e : \tau' & \\ \vec{\alpha} = FV(\tau') - FV(\Gamma) & (\Gamma - \Gamma(x)) \cup \{x : \forall\vec{\alpha}.\tau'\} \vdash e' : \tau\end{array}}{\Gamma \vdash (\lambda x\ .e')\ e : \tau}(\text{LET})$$

This set of inference rules represents an algorithm because there is exactly one conclusion for each syntactic ASL construct (giving priority to the (LET) rule over the regular application rule). This set of rules may be read as a Prolog program.

This algorithm has been proven to be:

- *syntactically sound*: when the algorithm succeeds on an expression e and returns a type τ, then $e : \tau$.
- *complete*: if an expression e possesses a type τ, then the algorithm will find a type τ' such that τ is an instance of τ'. The returned type τ' is thus the most general type of e.

Example We compute the type that we simply checked in our last example. What is drawn below is the result of the type synthesis: in fact, we run our algorithm with type variables representing unknowns, modifying the previous applications of the (INST) rule when necessary (i.e. when encountering an equality constraint). This is valid, since it can be proved that the correction of the whole deduction tree is preserved by substitution of types for type variables. In a real implementation of the algorithm, the data structures representing types will be submitted to a unification mechanism.

$$\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x : \alpha}\text{INST}_x}{\emptyset \vdash (\lambda x.x) : \alpha \to \alpha}\text{ABS}_x \quad \dfrac{\dfrac{}{\Gamma \vdash f\ : (\beta \to \beta) \to (\beta \to \beta)}\text{INST}_f \quad \dfrac{}{\Gamma \vdash f\ : \beta \to \beta}\text{INST}_f}{\Gamma = \{f : \forall\alpha.\alpha \to \alpha\} \vdash f\ f : \beta \to \beta}\text{APP}}{\emptyset \vdash (\lambda f.f\,f)(\lambda x.x) : \beta \to \beta}\text{LET}_f$$

Once again, this expression is not typable without the use of the (LET) rule: an error would occur because of the type equality constraints between all occurrences of a variable bound by a "λ". In an effective implementation, a unification error would occur.

□

We may notice from the above example that the algorithm is *syntax-directed*: we can reconstruct the ASL expression from its type deduction tree. From the above deduction tree, if we write upper rules as being "argument"s of the ones below and if we annotate the applications of the (INST) and (ABS) rules by the name of the subject variable , we obtain:

$$\text{LET}_f(\text{ABS}_x(\text{INST}_x),\ \text{APP}(\text{INST}_f,\ \text{INST}_f))$$

This is an illustration of the "types-as-propositions and programs-as-proofs" paradigm, also known as the "Curry-Howard isomorphism" (cf. [Howard, 1980]). In this example, we can see the type of the considered expression as a proposition and the expression itself as the proof, and, indeed, we recognise the expression as the deduction tree.

14.3 The ASL type-synthesizer

We now implement the set of inference rules given above.

We need:

- a CAML representation of ASL types and type schemes,
- a management of type environments,
- a unification procedure,
- a typing algorithm.

14.3.1 Representation of ASL types and type schemes

We first need to define a CAML type for our ASL type data structure:

```
#type asl_type = 'Unknown
#              | 'Number
#              | 'TypeVar of vartype
#              | 'Arrow of asl_type * asl_type
#and vartype = {Index:num; mutable Value:asl_type}
#and asl_type_scheme = 'Forall of num list * asl_type ;;
Type asl_type defined
     'Unknown : asl_type
   | 'Number : asl_type
   | 'TypeVar : (vartype -> asl_type)
   | 'Arrow : (asl_type * asl_type -> asl_type)
Type vartype defined
     .Index : (vartype -> num)
   ; .Value : (vartype -> asl_type)
Type asl_type_scheme defined
     'Forall : (num list * asl_type -> asl_type_scheme)
```

The `'Unknown` ASL type is not really a type: it is the initial value of fresh ASL type variables. We will consider as abnormal a situation where `'Unknown` appears in place of a regular ASL type. In such situations, we will raise the following exception:

```
#exception TypingBug of string;;
Exception TypingBug of string defined
```

Type variables are allocated by the `new_vartype` function, and their global counter (a local reference) is reset by `reset_vartypes`.

```
#let new_vartype, reset_vartypes =
#(* Generating and resetting unknowns *)
#    let counter = ref 0
#    in (function () -> counter:=!counter+1;
#                       {Index=!counter; Value='Unknown}),
```

```
#       (function () -> counter:=0; ());;
Value new_vartype = <fun> : (unit -> vartype)
Value reset_vartypes = <fun> : (unit -> unit)
```

14.3.2 Destructive unification of ASL types

We will need to "shorten" type variables: they are indirections to ASL types. This is needed for efficiency only.

```
#let rec shorten t =
#    match t with
#     'TypeVar {Index=_; Value='Unknown} -> t
#   | 'TypeVar ({Index=_;
#                Value='TypeVar {Index=_;
#                                Value='Unknown} as tv}) -> tv
#   | 'TypeVar ({Index=_; Value='TypeVar tv1} as tv2)
#            -> (tv2.Value <- tv1.Value); shorten t
#   | 'TypeVar {Index=_; Value=t'} -> t'
#   | 'Unknown -> raise TypingBug "shorten"
#   | t' -> t';;
Value shorten = <fun> : (asl_type -> asl_type)
```

An ASL type error will be represented by the following exception:

```
#exception TypeClash of asl_type * asl_type;;
Exception TypeClash of (asl_type * asl_type) defined
```

We will need unification on ASL types with *occur-check*. The following function implements occur-check:

```
#let occurs {Index=n;Value=_} = occrec
#  where rec occrec =
#  function 'TypeVar{Index=m;Value=_} -> (n=m)
#         | 'Number -> false
#         | 'Arrow(t1,t2) -> (occrec t1) or (occrec t2)
#         | 'Unknown -> raise TypingBug "occurs";;
Value occurs = <fun> : (vartype -> asl_type -> bool)
```

The unification function: implements destructive unification. Instead of returning the most general unifier, it returns the unificand of two types (their most general common instance). The two arguments are physically modified in order to represent the same type. The unification function will detect type clashes.

```
#let rec unify (tau1,tau2) =
#  match (shorten tau1, shorten tau2)
#  with (* type variable n and type variable m *)
#       ('TypeVar({Index=n; Value='Unknown} as tv1) as t1),
#       ('TypeVar({Index=m; Value='Unknown} as tv2) as t2)
```

```
#           -> if n=m then t2 else tv1.Value <- t2
#     | (* type t1 and type variable *)
#      t1, ('TypeVar ({Index=_;Value='Unknown} as tv) as t2)
#           -> if not(occurs tv t1) then tv.Value <- t1
#              else raise TypeClash (t1,t2)
#     | (* type variable and type t2 *)
#       ('TypeVar ({Index=_;Value='Unknown} as tv) as t1), t2
#           -> if not(occurs tv t2) then tv.Value <- t2
#              else raise TypeClash (t1,t2)
#     | 'Number, 'Number -> 'Number
#     | 'Arrow(t1,t2), ('Arrow(t'1,t'2) as t)
#           -> unify(t1,t'1); unify(t2,t'2); t
#     | other -> raise TypeClash other;;
Value unify = <fun> : (asl_type * asl_type -> asl_type)
```

14.3.3 Representation of typing environments

We use `asl_type_scheme list` as typing environments, and we will use the encoding of variables as numbers as indices into the environment.

The initial environment is a list of types `(Number -> (Number -> Number))`, which are the types of the ASL primitive functions.

```
#let init_typing_env =
#    map (function s ->
#           'Forall([],'Arrow('Number,
#                             'Arrow('Number,'Number))))
#        init_env;;
Value init_typing_env =
      [('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))))] :
      asl_type_scheme list
```

The global typing environment is initialised to the initial typing environment, and will be updated with the type of each ASL declaration, after they are type-checked.

```
#let global_typing_env = ref init_typing_env;;
Value global_typing_env =
      (ref
       [('Forall
```

```
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))));
       ('Forall
        ([],('Arrow ('Number,('Arrow ('Number,'Number))))))]) :
     asl_type_scheme list ref
```

14.3.4 From types to type schemes: generalisation

In order to implement generalisation, we will need some functions collecting types variables occurring in ASL types.

The following function computes the list of type variables of its argument.

```
#let vars_of_type tau = vars [] tau
# where rec vars vs =
#  function 'Number -> vs
#         | 'TypeVar {Index=n; Value='Unknown}
#                -> if mem n vs then vs else n::vs
#         | 'TypeVar {Index=_; Value= t} -> vars vs t
#         | 'Arrow(t1,t2) -> vars (vars vs t1) t2
#         | 'Unknown -> raise TypingBug "vars_of_type";;
Value vars_of_type = <fun> : (asl_type -> num list)
```

The `unknowns_of_type(bv,t)` application returns the list of variables occurring in `t` which do not appear in `bv`. The predefined `subtract` function returns the difference of two lists.

```
#subtract;;
<fun> : ('a list -> 'a list -> 'a list)

#let unknowns_of_type (bv,t) =
#    subtract (vars_of_type t) bv;;
Value unknowns_of_type = <fun> :
      (num list * asl_type -> num list)
```

We need to compute the list of unknowns of a type environment for the generalisation process (unknowns belonging to that list cannot become generic). The set of unknowns of a type environment is the union of the unknowns of each type. The `flat` function flattens a list of lists.

```
#flat;;
<fun> : ('a list list -> 'a list)

#let unknowns_of_type_env =
#    flat o (map (function 'Forall gv_t
```

```
#                           -> unknowns_of_type gv_t));;
Value unknowns_of_type_env = <fun> :
      (asl_type_scheme list -> num list)
```

The generalisation of a type is relative to a typing environment. The `make_set` function is predefined and eliminates duplications of its list argument.

```
#make_set;;
<fun> : ('a list -> 'a list)

#let generalise_type (gamma, tau) =
#  let genvars =
#        make_set (subtract (vars_of_type tau)
#                           (unknowns_of_type_env gamma))
#  in 'Forall(genvars, tau)
#;;
Value generalise_type = <fun> :
      (asl_type_scheme list * asl_type -> asl_type_scheme)
```

14.3.5 From type schemes to types: generic instanciation

The following function returns a generic instance of its type scheme argument. A generic instance is obtained by replacing all generic type variables by new unknowns:

```
#let gen_instance ('Forall(gv,tau)) =
#  (* We associate a new unknown to each generic variable *)
#  let unknowns =
#      map (function n -> n, 'TypeVar(new_vartype()))
#          gv
#  in ginstance tau
#  where rec ginstance = function
#        ('TypeVar {Index=n; Value='Unknown} as t) ->
#                     (try assoc n unknowns
#                      with failure "find" -> t)
#      | 'TypeVar {Index=_; Value= t} -> ginstance t
#      | 'Number -> 'Number
#      | 'Arrow(t1,t2) -> 'Arrow(ginstance t1, ginstance t2)
#      | 'Unknown -> raise TypingBug "gen_instance"
#;;
Value gen_instance = <fun> : (asl_type_scheme -> asl_type)
```

14.3.6 The ASL type synthesizer

The type synthesizer is the `asl_typing_expr` function. Each of its match cases corresponds to an inference rule given above.

```
#let rec asl_typing_expr gamma = type_rec
```

```
#where rec type_rec = function
#    'Const _ -> 'Number
#  | 'Var n ->
#      let sigma = (try nth gamma n  with failure _
#                          -> raise TypingBug "Unbound")
#      in gen_instance sigma
#  | 'Cond (e1,e2,e3) ->
#      let t1 = unify('Number, type_rec e1)
#      and t2 = type_rec e2 and t3 = type_rec e3
#      in unify(t2, t3); t3
#  | 'App(('Abs(x,e2) as f), e1) -> (* LET case *)
#      let t1 = type_rec e1 in
#      let sigma = generalise_type (gamma,t1)
#      in asl_typing_expr (sigma::gamma) e2
#  | 'App(e1,e2) ->
#      let u = 'TypeVar(new_vartype())
#      in unify(type_rec e1,'Arrow(type_rec e2,u)); u
#  | 'Abs(x,e) ->
#      let u = 'TypeVar(new_vartype()) in
#      let s = 'Forall([],u)
#      in 'Arrow(u,asl_typing_expr (s::gamma) e);;
Value asl_typing_expr = <fun> :
      (asl_type_scheme list -> asl -> asl_type)
```

14.3.7 Typing, trapping type clashes and printing ASL types

Now, we define some auxiliary functions in order to build a "good-looking" type synthesizer. First of all, a printing routine for ASL type schemes is defined (using a function `tvar_name` which computes a decent name for type variables).

```
#let tvar_name n =
# (* Computes a name "'a", ... for type variables, *)
# (* given an integer n representing the position  *)
# (* of the type variable in the list of generic   *)
# (* type variables                                *)
# let rec name_of n =
#    let q,r = (n quo 26), (n mod 26)
#    in if q=0 then ascii (96+r)
#       else (name_of q)^(ascii (96+r))
# in "'"^(name_of n)
#;;
Value tvar_name = <fun> : (num -> string)
```

Then a printing function for type schemes.

```
#let print_type_scheme ('Forall(gv,t)) =
# (* Prints a type scheme.              *)
```

```
# (* Fails when it encounters an unknown *)
# let names = (names_of (1,gv)
#      where rec names_of = function
#             (n,[]) -> []
#           | (n,(v1::Lv)) -> (tvar_name n)
#                             ::(names_of (n+1, Lv))) in
# let tvar_names = combine (rev gv,names)
# in print_rec t
#    where rec print_rec = function
#      'TypeVar{Index=n; Value='Unknown} ->
#        let name =
#            try assoc n tvar_names
#            with failure "find" ->
#              raise TypingBug "Non generic variable"
#        in print_string name
#    | 'TypeVar{Index=_;Value=t} -> print_rec t
#    | 'Number -> print_string "Number"
#    | 'Arrow(t1,t2) ->
#          print_string "("; print_rec t1;
#          print_string " -> "; print_rec t2;
#          print_string ")"
#    | 'Unknown -> raise TypingBug "print_type_scheme";;
Value print_type_scheme = <fun> : (asl_type_scheme -> unit)
```

Now, the main function which resets the type variables counter, calls the type synthesizer, traps ASL type clashes and prints the resulting types. At the end, the global environments are updated.

```
#let typing ('Decl(s,e)) =
# reset_vartypes();
# let tau = (* TYPING *)
#     try asl_typing_expr !global_typing_env e
#     with TypeClash(t1,t2) -> (* A typing error *)
#          let vars=vars_of_type(t1)@vars_of_type(t2) in
#          print_string "ASL Type clash between ";
#          print_type_scheme ('Forall(vars,t1));
#          print_string " and ";
#          print_type_scheme ('Forall(vars,t2));
#          raise failure "ASL typing"
#        | Unbound s -> raise TypingBug ("Unbound: "^s) in
# let sigma = generalise_type (!global_typing_env,tau) in
# (* UPDATING ENVIRONMENTS *)
# global_env := s::!global_env;
# global_typing_env := sigma::!global_typing_env;
# reset_vartypes ();
# (* PRINTING RESULTING TYPE *)
# print_string ("ASL Type of "^s^" is ");
```

```
# print_type_scheme sigma; print_newline();;
Value typing = <fun> : (top_asl -> unit)
```

14.3.8 Typing ASL programs

We reinitialise the parsing environment:

```
#global_env:=init_env; ();;
() : unit
```

Now let us run some examples of our ASL type checker:

```
#typing <<1;>>;;
ASL Type of it is Number
() : unit

#typing <<+ 2 ((\x.x) 3);>>;;
ASL Type of it is Number
() : unit

#typing <<if + 0 1 then 1 else 0 fi;>>;;
ASL Type of it is Number
() : unit

#typing <<let Id = \x.x;>>;;
ASL Type of Id is ('a -> 'a)
() : unit

#typing <<+ (Id 1) (Id Id 2);>>;;
ASL Type of it is Number
() : unit

#typing <<(\x.x x) (\x.x);>>;;
ASL Type of it is ('a -> 'a)
() : unit

#typing <<+ (\x.x) 1;>>;;
ASL Type clash between Number and ('a -> 'a)
Evaluation Failed: ASL typing
```

14.3.9 Typing and recursion

The Z fixpoint combinator does not have a type in Milner's type system:

```
#typing <<let Z= \f.((\x.f(\z.(x x)z))
#                    (\x.f(\z.(x x)z)));>>;;
ASL Type clash between 'a and ('a -> 'b)
Evaluation Failed: ASL typing
```

This is because we try to apply `x` to itself, and the type of `x` is not polymorphic. In fact, no fixpoint combinator is typable in ASL. This is why we need a special primitive or syntactic construct in order to express recursivity.

If we want to assign types to recursive programs, we have to predefine the Z fixpoint combinator. Its type scheme should be $\forall\alpha.((\alpha \to \alpha) \to \alpha)$, because we take fixpoints of functions.

```
#global_env:="Z"::init_env;
#global_typing_env:=
#    ('Forall([1],
#     'Arrow('Arrow('TypeVar{Index=1;Value='Unknown},
#                   'TypeVar{Index=1;Value='Unknown}),
#            'TypeVar{Index=1;Value='Unknown})))
#   ::init_typing_env;
#();;
() : unit
```

We may now define our favorite functions as:

```
#typing
#  <<let fact =
#          Z (\f.(\n. if = n 0 then 1
#                     else * n (f (- n 1))
#                     fi));>>;;
ASL Type of fact is (Number -> Number)
() : unit

#typing
#  <<let fib =
#         Z (\f.(\n. if = n 1 then 1
#                    else if = n 2 then 1
#                         else + (f (- n 1)) (f (- n 2))
#                         fi
#                    fi));>>;;
ASL Type of fib is (Number -> Number)
() : unit

#typing <<fact 25;>>;;
ASL Type of it is Number
() : unit

#typing <<fib 10;>>;;
ASL Type of it is Number
() : unit
```

Chapter 15

Compiling ASL to an abstract machine code

In order to fully take advantage of the static typing of ASL programs, we have to:

- either write a new interpreter without type tests (hard because we used pattern-matching in order to realise type tests);
- or design an untyped machine and produce code for it.

We choose here the second solution: it will permit us to give some intuition about the compiling process of functional languages, and to describe a typical execution model for (strict) functional languages. The machine that we will use is a simplified version the *Categorical Abstract Machine* (CAM, for short). The CAM is the CAML execution model.

We will call CAM our abstract machine, despite its differences with the original CAM. For more informations on the CAM, see [Cousineau *et al.*, 1985] or [Mauny and Suárez, 1986].

15.1 The Abstract Machine

The execution model is a *stack machine* (i.e. a machine using a stack). In this section, we define in CAML the *states* of the CAM and its instructions.

A state is composed of:

- a *register* (holding values and environments),
- a *program counter*, represented here as a list of instructions whose first element is the current instruction being executed,
- and a *stack* represented as a list of code addresses (instruction lists), values and environments.

The first CAML type that we need is the type for CAM instructions. We will study later the effect of each instruction.

In the following type definition, the CAML system issues a warning message concerning a **re**definition of the type `instruction`. The reason is that a type `instruction` already exists: this one will no longer be accessible.

```
#type instruction =
#     'Quote of num    (* Numerical constants *)
#                      (* Arithmetic operations *)
#   | 'Plus | 'Minus | 'Times | 'Divide | 'Equal
#   | 'Nth of num      (* Variable accesses *)
#   | 'Branch of instruction list * instruction list
#                      (* Conditional execution *)
#   | 'Push            (* Pushes onto the stack *)
#   | 'Swap            (* Exch. top of stack and register *)
#   | 'Clos of instruction list
#                      (* Builds a closure with the current
#                                              environment *)
#   | 'Apply           (* Function application *);;
Warning: type instruction redefined
Type instruction defined
     'Quote : (num -> instruction)
   | 'Plus : instruction
   | 'Minus : instruction
   | 'Times : instruction
   | 'Divide : instruction
   | 'Equal : instruction
   | 'Nth : (num -> instruction)
   | 'Branch :
     (instruction list * instruction list -> instruction)
   | 'Push : instruction
   | 'Swap : instruction
   | 'Clos : (instruction list -> instruction)
   | 'Apply : instruction
```

We need a new type for semantic values since instruction lists have now replaced abstract syntax trees. The semantic values are merged in a type `object`. The type `object` behaves as data in a computer memory: we need higher-level information (such as type information) in order to interpret them. Furthermore, some data do not correspond to anything (for example an environment composed of environments represents neither an ASL value nor an intermediate data in a legal computation process).

```
#type object = 'Constant of num
#            | 'Closure of object * object
#            | 'Address of instruction list
#            | 'Environment of object list
#;;
Type object defined
     'Constant : (num -> object)
   | 'Closure : (object * object -> object)
   | 'Address : (instruction list -> object)
   | 'Environment : (object list -> object)
```

The type `state` is a product type with mutable components.

```
#type state = {mutable Reg: object;
#              mutable PC: instruction list;
#              mutable Stack: object list}
#;;
Type state defined
     .Reg : (state -> object)
   ; .PC : (state -> instruction list)
   ; .Stack : (state -> object list)
```

Now, we give the *operational semantics* of CAM instructions. The effect of an instruction is to change the state configuration. This is what we describe now with the `CAMstep` function. Code executions will be arbitrary iterations of this function.

```
#exception CAMbug of string;;
Exception CAMbug of string defined

#exception CAM_End of object;;
Exception CAM_End of object defined

#let CAMstep S = match S with
#  {Reg=_; PC='Quote(n)::C; Stack=s} ->
#                (S.Reg <- 'Constant(n); S.PC <- C)
#
#| {Reg='Constant(m); PC='Plus::C; Stack='Constant(n)::s} ->
#                (S.Reg <- 'Constant(n+m);
#                 S.Stack <- s; S.PC <- C)
#
#| {Reg='Constant(m); PC='Minus::C; Stack='Constant(n)::s} ->
#                (S.Reg <- 'Constant(n-m);
#                 S.Stack <- s; S.PC <- C)
#
#| {Reg='Constant(m); PC='Times::C; Stack='Constant(n)::s} ->
#                (S.Reg <- 'Constant(n*m);
#                 S.Stack <- s; S.PC <- C)
#
#| {Reg='Constant(m); PC='Divide::C; Stack='Constant(n)::s} ->
#                (S.Reg <- 'Constant(n/m);
#                 S.Stack <- s; S.PC <- C)
#
#| {Reg='Constant(m); PC='Equal::C; Stack='Constant(n)::s} ->
#                (S.Reg <- 'Constant( if n=m then 1 else 0);
#                 S.Stack <- s; S.PC <- C)
#
#| {Reg='Constant(m); PC='Branch(C1,C2)::C; Stack=r::s} ->
```

```
#                (S.Reg <- r;
#                 S.Stack <- 'Address(C)::s;
#                 S.PC <- if m=0 then C2 else C1)
#
#| {Reg=r; PC='Push::C; Stack=s} ->
#                (S.Stack <- r::s; S.PC <- C)
#
#| {Reg=r1; PC='Swap::C; Stack=r2::s} ->
#                (S.Reg <- r2; S.Stack <- r1::s; S.PC <- C)
#
#| {Reg=r; PC='Clos(C1)::C; Stack=s} ->
#                (S.Reg <- 'Closure('Address(C1),r);
#                 S.PC <- C)
#
#| {Reg=_; PC=[]; Stack='Address(add)::s} ->
#                (S.Stack <- s; S.PC <- add)
#
#| {Reg=v; PC='Apply::C;
#          Stack='Closure('Address(C1),'Environment(e))::s} ->
#                (S.Reg <- 'Environment(v::e);
#                 S.Stack <- 'Address(C)::s; S.PC <- C1)
#
#| {Reg=v; PC=[]; Stack=[]} -> raise CAM_End v
#| {Reg=_; PC=('Plus|'Minus|'Times|'Divide|'Equal)::C;
#          Stack=_::_} -> raise CAMbug "IllTyped"
#| {Reg='Environment(e); PC='Nth(n)::C; Stack=_} ->
#                (S.Reg <- (try nth e n
#                           with failure _ ->
#                             raise CAMbug "IllTyped");
#                 S.PC <- C)
#| _ -> raise CAMbug "Wrong configuration";;
Value CAMstep = <fun> : (state -> instruction list)
```

We may notice that the empty code sequence denotes a (possibly final) *return* instruction.

We could argue that pattern-matching in the `CAMLstep` function implements a kind of dynamic typechecking. In fact, in a concrete (low-level) implementation of the machine (expansion of the CAM instructions in assembly code, for example), these tests would not appear. They are useless since we trust the typechecker and the compiler. So, any execution error in a real implementation comes from a *bug* in one of the above processes and would lead to memory errors or illegal instructions (usually detected by the computer's operating system).

15.2 Compiling ASL programs into CAM code

We give in this section a compiling function taking the abstract syntax tree of an ASL expression and producing CAM code. The compilation scheme is very simple:

- the code of a constant is `Quote`;
- a variable is compiled as an access to the appropriate component of the current environment (`Nth`);
- the code of a conditional expression will save the current environment (`Push`), evaluate the condition part, and, according to the boolean value obtained, select the appropriate code to execute (`Branch`);
- the code of an application will also save the environment on the stack (`Push`), execute the function part of the application, then exchange the functional value and the saved environment (`Swap`), evaluate the argument and, finally, apply the functional value (which is at the top of the stack) to the argument held in the register with the `Apply` instruction;
- the code of an abstraction simply consists in building a closure representing the functional value: the closure is composed of the code of the function and the current environment.

Here is the compiling function:

```
#let rec code_of = function
#  'Const(n) -> ['Quote(n)]
#| 'Var n -> ['Nth(n)]
#| 'Cond(e_test,e_t,e_f) ->
#         'Push::(code_of e_test)
#       @['Branch(code_of e_t, code_of e_f)]
#| 'App(e1,e2) -> 'Push::(code_of e1)
#                @['Swap]@(code_of e2)
#                @['Apply]
#| 'Abs(_,e) -> ['Clos(code_of e)];;
Value code_of = <fun> : (asl -> instruction list)
```

A global environment is needed in order to maintain already defined values. Any CAM execution will start in a state whose register part contains this global environment.

```
#let init_CAM_env =
#  let basic_instruction = function
#        "+" -> 'Plus
#      | "-" -> 'Minus
#      | "*" -> 'Times
#      | "/" -> 'Divide
#      | "=" -> 'Equal
#      | s -> raise CAMbug ("Unknown primitive "^s)
#  in map (function s ->
#           'Closure('Address['Clos('Push::'Nth(2)
#                                  ::'Swap::'Nth(1)
#                                  ::[basic_instruction s])],
#                    'Environment []))
#         init_env;;
```

```
Value init_CAM_env =
      [('Closure
        (('Address
          [('Clos ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Plus])]),
         ('Environment [])));
       ('Closure
        (('Address
          [('Clos
            ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Minus])]),
         ('Environment [])));
       ('Closure
        (('Address
          [('Clos
            ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Times])]),
         ('Environment [])));
       ('Closure
        (('Address
          [('Clos
            ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Divide])]),
         ('Environment [])));
       ('Closure
        (('Address
          [('Clos
            ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Equal])]),
         ('Environment [])))] :
      object list

#let global_CAM_env = ref init_CAM_env;;
Value global_CAM_env =
      (ref
       [('Closure
         (('Address
           [('Clos
             ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Plus])]),
          ('Environment [])));
        ('Closure
         (('Address
           [('Clos
             ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Minus])]),
          ('Environment [])));
        ('Closure
         (('Address
           [('Clos
             ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Times])]),
          ('Environment [])));
```

```
        ('Closure
         (('Address
           [('Clos
             ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Divide])]),
          ('Environment [])));
        ('Closure
         (('Address
           [('Clos
             ['Push; ('Nth 2); 'Swap; ('Nth 1); 'Equal])]),
          ('Environment []))))]) :
      object list ref
```

As an example, here is the code for some ASL expressions:

```
#code_of <:asl:expr<1>>;;
[('Quote 1)] : instruction list

#code_of <:asl:expr<+ 1 2>>;;
['Push; 'Push; ('Nth 6); 'Swap; ('Quote 1); 'Apply; 'Swap;
 ('Quote 2); 'Apply] : instruction list

#code_of <:asl:expr<(\x.x) ((\x.x) 0)>>;;
['Push; ('Clos [('Nth 1)]); 'Swap; 'Push;
 ('Clos [('Nth 1)]); 'Swap; ('Quote 0); 'Apply; 'Apply] :
 instruction list

#code_of <:asl:expr<1 + if 0 then 2 else 3 fi>>;;
['Push; 'Push; ('Quote 1); 'Swap; ('Nth 6); 'Apply; 'Swap;
 'Push; ('Quote 0); ('Branch ([('Quote 2)],[('Quote 3)]));
 'Apply] : instruction list
```

15.3 Execution of CAM code

The main function for executing compiled ASL manages the global environment until execution has succeeded.

```
#let ASL_run ('Decl(s,e)) =
#  (* TYPING *)
#    reset_vartypes();
#    let tau =
#        try asl_typing_expr !global_typing_env e
#        with TypeClash(t1,t2) ->
#              let vars=vars_of_type(t1) @ vars_of_type(t2) in
#              print_string "ASL Type clash between ";
#              print_type_scheme ('Forall(vars,t1));
#              print_string " and ";
```

```
#              print_type_scheme ('Forall(vars,t2));
#              raise failure "ASL typing"
#           | Unbound s -> raise TypingBug ("Unbound: "^s) in
#    let sigma = generalise_type (!global_typing_env,tau) in
#  (* PRINTING TYPE INFORMATION *)
#    print_string "ASL Type of ";
#    print_string s; print_string " is ";
#    print_type_scheme sigma; print_newline();
#  (* COMPILING *)
#    let C = code_of e in
#    let S = {Reg='Environment(!global_CAM_env);
#             PC=C; Stack=[]} in
#  (* EXECUTING *)
#    let result = try while true do CAMstep S done; S.Reg
#                 with CAM_End v -> v in
#  (* UPDATING ENVIRONMENTS *)
#    global_env := s::!global_env;
#    global_typing_env := sigma::!global_typing_env;
#    global_CAM_env := result::!global_CAM_env;
#  (* PRINTING RESULT *)
#    (match result
#     with 'Constant(n) -> print_num n
#        | 'Closure(_) -> print_string "<funval>"
#        | _ -> raise CAMbug "Wrong state configuration");
#    print_newline();;
Value ASL_run = <fun> : (top_asl -> unit)
```

Now, let us run some examples:

```
#(* Reinitialising environments *)
#global_env:=init_env;
#global_typing_env:=init_typing_env;
#global_CAM_env:=init_CAM_env;
#();;
() : unit

#ASL_run <<1;>>;;
ASL Type of it is Number
1
() : unit

#ASL_run <<+ 1 2;>>;;
ASL Type of it is Number
3
() : unit
```

```
#ASL_run <<(\f.(\x.f x)) (\x. + x 1) 3;>>;;
ASL Type of it is Number
4
() : unit
```

We may now introduce the Z fixpoint combinator as a predefined function.

```
#(* Introducing the Z fixpoint combinator *)
#global_env:="Z"::init_env;
#global_typing_env:=
#  ('Forall([1],
#           'Arrow('Arrow('TypeVar{Index=1;Value='Unknown},
#                         'TypeVar{Index=1;Value='Unknown}),
#                  'TypeVar{Index=1;Value='Unknown})))
#::init_typing_env;
#global_CAM_env:=
#  (match code_of <:asl:expr<\f.((\x.f(\z.(x x)z))
#                                (\x.f(\z.(x x)z)))>>
#   with ['Clos(C)] -> 'Closure('Address(C), 'Environment [])
#      | _ -> raise CAMbug "Wrong code for Z")
# ::init_CAM_env; ();;
() : unit

#ASL_run <<Z;>>;;
ASL Type of it is (('a -> 'a) -> 'a)
<funval>
() : unit

#ASL_run <<let fact =  Z (\f.(\n. if = n 0 then 1
#                                 else * n (f (- n 1))
#                                 fi));>>;;
ASL Type of fact is (Number -> Number)
<funval>
() : unit

#ASL_run <<let fib =
#         Z (\f.(\n. if = n 1 then 1
#                    else if = n 2 then 1
#                         else + (f (- n 1)) (f (- n 2))
#                                  fi
#                    fi));>>;;
ASL Type of fib is (Number -> Number)
<funval>
() : unit

#ASL_run <<fact 25;>>;;
```

```
ASL Type of it is Number
155112100433309859840000000
() : unit

#ASL_run <<fib 10;>>;;
ASL Type of it is Number
55
() : unit
```

It is of course possible (and desirable) to introduce recursion by using a specific syntactic construct, special instructions and a dedicated case to the compiling function. See [Mauny and Suárez, 1986] for efficient compilation of recursion, data structures etc.

Chapter 16

Answers to exercises

We give in this chapter one possible solution for each exercise contained in this document. Excercises are referred by their number and the page where they have been proposed: for example, "2.1, p. 15" refers to the first exercise in chapter 2; this exercise is located on page 15.

3.1, p. 15

```
#(function f -> (function m -> f(m+1) / 2));;
<fun> : ((num -> num) -> num -> num)

#function m -> (function n -> n+m+1);;
<fun> : (num -> num -> num)

#function f -> (f 2)+1;;
<fun> : ((num -> num) -> num)
```

3.2, p. 15

We must first rename `y` in `z`, obtaining:

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

and finally:

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

Without renaming, we would have obtained:

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

which does not denote the same function.

3.3, p. 15

We write successively the reduction steps for each expressions, and then we use CAML in order to check our result.

- ```
  let x=1+2 in ((function y -> y+x) x);;
  (function y -> y+(1+2)) (1+2);;
  (function y -> y+(1+2)) 3;;
  3+(1+2);;
  3+3;;
  6;;
  ```

  CAML says:

  ```
  #let x=1+2 in ((function y -> y+x) x);;
  6 : num
  ```

- ```
  let x=1+2 in ((function x -> x+x) x);;
  (function x -> x+x) (1+2);;
  3+3;;
  6;;
  ```

  CAML says:

  ```
  #let x=1+2 in ((function x -> x+x) x);;
  6 : num
  ```

- ```
  let f1 = function f2 -> (function x -> f2 x)
      in let g = function x -> x+1
             in f1 g 2;;
  let g = function x -> x+1
      in function f2 -> (function x -> f2 x) g 2;;
  (function f2 -> (function x -> f2 x)) (function x -> x+1) 2;;
  (function x -> (function x -> x+1) x)  2;;
  (function x -> x+1) 2;;
  2+1;;
  3;;
  ```

  CAML says:

  ```
  #let f1 = function f2 -> (function x -> f2 x)
  #    in let g = function x -> x+1
  #           in f1 g 2;;
  3 : num
  ```

4.1, p. 26

This exercise can be very simple—follow your inspiration!. The computation of the surface of a rectangle could be expresed as:

```
#let sq_rect (l,L) = l*L;;
Value sq_rect = <fun> : (num * num -> num)
```
```

4.2, p. 26

In a call-by-value language without conditional construct (and without sum types), all programs involving a recursive definition fail to terminate.

4.3, p. 26

```
#let rec factorial n = if n=1 then 1 else n*(factorial(n-1));;
Value factorial = <fun> : (num -> num)

#factorial 5;;
120 : num

#let factorial n = fact n 1
#where rec fact n m = if n=1 then m else fact (n-1) (n*m);;
Value factorial = <fun> : (num -> num)

#factorial 5;;
120 : num
```

4.4, p. 26

```
#let rec fibonacci n = if n=1 then 1
#                      else if n=2 then 1
#                           else fibonacci(n-1) + fibonacci(n-2);;
Value fibonacci = <fun> : (num -> num)

#fibonacci 20;;
6765 : num
```

4.5, p. 26

```
#curry (prefix o);;
<fun> : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)

#curry o uncurry;;
<fun> : (('a -> 'b -> 'c) -> 'a -> 'b -> 'c)

#uncurry o curry;;
<fun> : (('a * 'b -> 'c) -> 'a * 'b -> 'c)
```

5.1, p. 30

```
#let rec combine =
#  function [],[] -> []
#         | (x1::L1),(x2::L2) -> (x1,x2)::(combine(L1,L2))
#         | _ -> raise failure "combine: lists of different length";;
```

```
Value combine = <fun> :
      ('a list * 'b list -> ('a * 'b) list)

#combine ([1;2;3],["a";"b";"c"]);;
[(1,"a"); (2,"b"); (3,"c")] : (num * string) list

#combine ([1;2;3],["a";"b"]);;

Evaluation Failed: combine: lists of different length
```

5.2, p. 30

```
#let rec sublists =
#    function [] -> [[]]
#           | x::L -> let SL = sublists L
#                     in SL @ (map (fun l -> x::l) SL);;
Value sublists = <fun> : ('a list -> 'a list list)

#sublists [];;
[[]] : 'a list list

#sublists [1;2;3];;
[[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]] :
 num list list

#sublists ["a"];;
[[]; ["a"]] : string list list
```

6.1, p. 41

```
#let rec nodes_and_leaves =
#    function 'Leaf x -> ([],[x])
#           | 'Btree {Op=x; Son1=s1; Son2=s2} ->
#               let (nodes1,leaves1) = nodes_and_leaves s1
#               and (nodes2,leaves2) = nodes_and_leaves s2
#               in (x::nodes1@nodes2, leaves1@leaves2);;
Value nodes_and_leaves = <fun> :
      (('a,'b) btree -> 'a list * 'b list)

#nodes_and_leaves ('Btree {Op="+"; Son1='Leaf 1; Son2='Leaf 2});;
(["+"],[1; 2]) : (string list * num list)
```

6.2, p. 41

```
#let map_btree f g = map_rec
#where rec map_rec =
```

```
#       function 'Leaf x -> 'Leaf (f x)
#              | 'Btree {Op=op; Son1=s1; Son2=s2} ->
#                'Btree {Op=g op; Son1=map_rec s1; Son2=map_rec s2};;
Value map_btree = <fun> :
      (('a -> 'b) -> ('c -> 'd) -> ('c,'a) btree ->
       ('d,'b) btree)
```

6.3, p. 42

We need to give a functional interpretation to `btree` data constructors. We use `f` (resp. `g`) to denote the function associated to the `'Leaf` (resp. `'Btree`) data constructor, obtaining the following CAML definition:

```
#let btree_iterate bt f g = it_rec bt
#where rec it_rec =
#       function 'Leaf x -> f x
#              | 'Btree{Op=op; Son1=s1; Son2=s2} ->
#                       g op (it_rec s1) (it_rec s2);;
Value btree_iterate = <fun> :
      (('a,'b) btree -> ('b -> 'c) ->
       ('a -> 'c -> 'c -> 'c) -> 'c)

#btree_iterate ('Btree {Op="+"; Son1='Leaf 1; Son2='Leaf 2})
#              (function x -> x)
#              (function "+" -> add
#                      | _ -> raise failure "Unknown op");;
3 : num
```

7.1, p. 52

We use the parameterized type `stream`, the functions `map_stream`, `nth_stream` and the stream of natural numbers `nats`.

```
#nats;;
{Hd=0; Tl=*} : num stream

#let nats_from_2 = nats.Tl.Tl;;
Value nats_from_2 = {Hd=2; Tl=*} : num stream

#let rec filter_stream p {Hd=n; Tl=s} =
#       if (n mod p) = 0 then filter_stream p s
#       else sieve {Hd=n; Tl=filter_stream p s}
#and sieve {Hd=n;Tl=s} = {Hd=n; Tl=filter_stream n s};;
Value filter_stream = <fun> :
      (num -> num stream -> num stream)
Value sieve = <fun> : (num stream -> num stream)
```

```
#let primes = sieve nats_from_2;;
Value primes = {Hd=2; Tl=*} : num stream

#nth_stream primes 10;;
31 : num

#primes;;
{Hd=2;
 Tl=
  {Hd=3;
   Tl=
    {Hd=5;
     Tl=
      {Hd=7;
       Tl=
        {Hd=11;
         Tl=
          {Hd=13;
           Tl=
            {Hd=17;
             Tl=
              {Hd=19;
               Tl={Hd=23; Tl={Hd=29; Tl={Hd=31; Tl=*}}}}}}}}}} :
 num stream
```

7.2, p. 52

Not given.

7.3, p. 52

Not given.

7.4, p. 52

```
#type ('a,'b) lisp_cons = {mutable Car:'a; mutable Cdr:'b};;
Type lisp_cons defined
     .Car : (('a,'b) lisp_cons -> 'a)
   ; .Cdr : (('b,'a) lisp_cons -> 'a)

#let car p = p.Car
#and cdr p = p.Cdr
#and rplaca p v = p.Car <- v
#and rplacd p v = p.Cdr <- v;;
Value car = <fun> : (('a,'b) lisp_cons -> 'a)
Value cdr = <fun> : (('a,'b) lisp_cons -> 'b)
```

```
Value rplaca = <fun> : (('a,'b) lisp_cons -> 'a -> 'a)
Value rplacd = <fun> : (('a,'b) lisp_cons -> 'b -> 'b)

#let p = {Car=1; Cdr=true};;
Value p = {Car=1; Cdr=true} : (num,bool) lisp_cons

#rplaca p 2;;
2 : num

#p;;
{Car=2; Cdr=true} : (num,bool) lisp_cons
```

8.1, p. 56

```
#let find_succeed f = find_rec
#where rec find_rec =
#        function [] -> raise failure "find_succeed"
#               | x::L -> try f x; x with _ -> find_rec L;;
Value find_succeed = <fun> : (('a -> 'b) -> 'a list -> 'a)

#find_succeed hd [[];[];[1;2];[3;4]];;
[1; 2] : num list
```

8.2, p. 56

```
#let map_succeed f = map_f
#where rec map_f =
#        function [] -> []
#               | h::t -> try (f h)::(map_f t)
#                         with _ ->  map_f t;;
Value map_succeed = <fun> :
      (('a -> 'b) -> 'a list -> 'b list)

#map_succeed hd [[];[1];[2;3];[4;5;6]];;
[1; 2; 4] : num list
```

Chapter 17

Conclusion and further reading

We have not been exhaustive in the description of the CAML features. We only introduced general concepts in functional programming, and we have insisted on the features used in the prototyping of ASL. Further information about CAML can be found in [Cousineau and Huet, 1990] and [Weis *et al.*, 1990]. [Hardin and Viguié, 1991] contains also an introduction to functional programming in CAML.

We still hope to have convinced the reader of the "clean" approach to programming provided by functional languages.

More information on unification and operations on first-order terms can be found in [Huet, 1986]. An introduction to lambda-calculus and type systems can be found in [Huet, 1990], [Krivine, 1990] and [Hindley and Seldin, 1986].

The description of the implementation of call-by-value functional programming languages can be found in [Leroy, 1990].

The implementation of lazy functional languages is described in [Peyton-Jones, 1987][1]. An introduction to programming in lazy functional languages can be found in [Bird and Wadler, 1986].

[1] of which [Peyton-Jones, 1990] is a French translation.

Bibliography

[Backus, 1978] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. In *Communications of the ACM*, volume 21, pages 133–140, 1978.

[Bird and Wadler, 1986] R. Bird and P. Wadler. *Introduction to Functional Programming.* Series in Computer Science. Prentice-Hall International, 1986.

[Church, 1941] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Clément *et al.*, 1986] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[Cousineau and Huet, 1990] G. Cousineau and G. Huet. The CAML primer. Technical Report 122, INRIA, 1990.

[Cousineau *et al.*, 1985] G. Cousineau, P.-L. Curien, and M. Mauny. *The Categorical Abstract Machine*, pages 50–64. Number 201 in Lecture Notes in Computer Science. Springer Verlag, 1985.

[De Bruijn, 1962] N. De Bruijn. *Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation.* Indag. Math., 1962.

[Gordon *et al.*, 1978] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadworth. A metalanguage for interactive proofs in LCF. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 119–130, 1978.

[Hardin and Viguié, 1991] T. Accart Hardin and V. Donzeau-Gouge Viguié. *Apprentissage de la Programmation avec CAML et Ada.* InterEditions, 1991. To appear.

[Hindley and Seldin, 1986] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society, Student Texts. Cambridge University Press, 1986.

[Howard, 1980] W.A. Howard. *The formulae-as-type notion of construction*, pages 479–490. Academic Press, 1980.

[Hudak and Wadler, 1990] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, 1990.

[Huet, 1986] G. Huet. Formal structures for computation and deduction. Course Notes, 1986.

[Huet, 1990] G. Huet. Initiation au lambda-calcul. Notes de cours, 1990.

[Johnson, 1986] S.C. Johnson. Yacc: Yet Another Compiler-Compiler. In *Unix Programmer's Manual*, volume PS1. Usenix Association, 1986.

[Johnsson, 1984] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of ACM Conference on Compiler Construction*, pages 58–69, 1984.

[Krivine, 1990] J.-L. Krivine. *Lambda-calcul, types et modèles.* Etudes et recherches en informatique. Masson, 1990.

[Landin, 1966] P. Landin. The next 700 programming languages. In *Communications of the ACM*, volume 9, pages 157–164, 1966.

[Leroy and Mauny, 1991] X. Leroy and M. Mauny. Dynamics in ML. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[Leroy, 1990] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[MacCarthy, 1962] J. MacCarthy. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.

[MacQueen, 1984] D. MacQueen. Modules for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1984.

[Mauny and Suárez, 1986] M. Mauny and A. Suárez. Implementing functional languages in the categorical abstract machine. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 266–278, 1986.

[Mauny, 1989] M. Mauny. Parsers and printers as stream destructors and constructors embedded in functional languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1989.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[Milner, 1987] R. Milner. A proposal for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1987.

[Peyton-Jones, 1987] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, 1987.

[Peyton-Jones, 1990] S.L. Peyton-Jones. *Mise en œuvre des langages fonctionnels de programmation.* Manuels Informatiques Masson. Masson, 1990.

[Robinson, 1971] J. A. Robinson. Computational logic: the unification computation. In *Machine Intelligence*, volume 6 of *Americal Elsevier*. B. Meltzer and D. Mitchie (Eds), 1971.

[Turner, 1976] D. Turner. SASL language manual. Technical report, St Andrews University, 1976.

[Turner, 1982] D. Turner. *Recursion equations as a programming language*, pages 1–28. Cambridge University Press, 1982.

[Turner, 1985] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer Verlag, 1985.

[Weis *et al.*, 1990] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.