Functional Programming with Static Typing in Caml

Opposite to Scheme and LISP, based on latent treatment of types, there are programming languages – in particular functional programming languages – where the type system is elaborated and sound. In these languages, backed theoretically by the Typed Lambda Calculus, type checking adopts an interesting solution: instead of verifying the processing validity of the different program objects according to their explicitly declared types, the types are automatically inferred starting from the way the objects are processed. Such languages are statically typed (i.e. type checking is performed at compile time). Moreover, correct data processing is enforced through the rules of a mathematically well founded type system.

By a *type system* we mean a set of rules and mechanisms used in a programming language to organize, build and handle the types accepted in the language. These mechanisms and rules address the following major aspects:

- Defining new types.
- Associating types with various language constructs.
- Deciding on *type equivalence* i.e., determine when distinct types are the same.
- Checking *type compatibility* i.e., find out if a value of a given type can be correctly used in a given processing context.
- *Inferring the types* of the language constructs when they are not explicitly declared i.e., apply rules for synthesizing the type of a construct starting from the types of its components.

As a base for the discussion of the above-mentioned problems addressing type systems and type processing, we are using a member of the ML family languages: Caml. Almost all that has been discussed for Scheme applies to Caml as well. As in Scheme, the basic element of the language is the function, considered as a first class value. An **1**-ary function, is represented as

fun *formal* -> *expression*

and, as in Scheme, produces a functional closure. However, Caml is a fully statically scoped language and the top-level variables have themselves static scope: the part of the program following their point of definition. This helps observing referential transparency, if side effects are not abused. The top-level special expressions

```
let variable = expression;;
let rec variable = expression;;
```

are similar to the **define** top-level binding expression in Scheme. They bind the ***variable*** to the value of the ***expression***. The scope, or region, of the ***variable*** is the part of the program that follows the **let** expression and, respectively, the part of the program that follows the **let rec** expression including the **let rec** itself. For instance, the first expression below binds the variable **plus** to a functional closure (a curried function). The second expression binds **factorial** to a recursive function that computes **n!**.

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

let rec factorial = fun n -> if n=1 then 1 else n*factorial(n-1);;
factorial: int -> int = <fun>
```

Notice that an expression produces two values: the value of the expression and the type of the value. We shall call *signature* the type of a function. For instance, the value of **plus** is a function, designated by **<fun>**, the domain of which is **int** and the range **int -> int**, i.e.

the signature corresponding to functions whose domain and range are **int**. The type of an expression need not be explicitly specified; it is inferred based on the types of its components. Type inference implies type checking as a natural subtask.

As far as the safety of the recursive definition of **factorial** is concerned, traps as those form Scheme could not appear. The variable **factorial** is saved within the functional closure that is bound to it and it is statically scoped.

The application of a function follows the applicative-order evaluation: parameters are passed by value. The syntax of function application is more permissive **(f p)** is equivalent to **f(p)** and to **f p**. If **f** is an **n**-ary curried function, then its application on the first **k** parameters must be written either **(f p1 … pk)** or **f p1 … pk** (if this last format does not create ambiguities). In addition, an **n**-ary curried function can be defined as **fun *p1 p2 … pn* -> *expression***, similarly to the convention in the λ_0 language. The function **plus** can be written:

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

In addition, a more convenient short-cut notation exists:

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

Apart from Scheme, the functions can be used both in infix or prefix form. There is a predefined function, called **prefix**, which transforms an infix function into an equivalent prefix curried function. For example, the application **1 + 2** is equivalent to **(prefix +) 1 2**. In addition, there is a directive that helps defining infix functions, such as:

```
#infix "o";;
let (prefix o) = fun f -> fun g -> fun x -> f (g x);;
o: ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>

let ff = (fun x -> x*x) o (fun x -> x+x);;
ff: int -> int = <fun>

ff 2;;
-: int = 16
```

In the example above, the types in the signature of the composition function **o** are not constants. They are generic types represented by type variables in the format ***'identifier***. The function **o** is polymorphic. It can take arguments of any type that obey its signature.

As in Scheme, there are scoping expressions. The **let** and **letrec** from Scheme have similar equivalents in Caml.

let var_1 **=** $expr_1$ **and** var_2 **=** $expr_2$ **and … and** var_n **=** $expr_n$ **in** ***expr***

The scope of var_i **i=1,n** is the expression ***expr***. The expressions $expr_i$ are evaluated in an unspecified order. The variable var_i is bound to the result of evaluating $expr_i$ and then ***expr*** is evaluated. The result of ***expr*** is the result of the **let** expression.

let rec var_1 **=** $expr_1$ **and** var_2 **=** $expr_2$ **and … and** var_n **=** $expr_n$ **in** ***expr***

The expression is similar to **let** with the difference that the scope of var_i is the entire **let rec** expresssion. In other words, the expressions $expr_i$ can refer the variables var_i .

When defining top-level variables **let** and **let rec** are equivalent to:

```
let var = expr in the_rest_of_the_program
let rec var = expr in the_rest_of_the_program
```

```
let is_even =
    let rec is_even = fun n -> n = 0 or n > 0 & is_odd(n-1)
        and is_odd  = fun n -> n = 1 or n > 1 & is_even(n-1)
    in is_even;;
is_even: int -> bool = <fun>
```

The operator **&** is the Boolean **and**; it is non-strict. The value of the top-level variable **is_even** is the functional closure corresponding to the local variable **is_even**. The closure saves the local variables **is_even** and **is_odd**. In other words, the closure saves the auxiliary functions.

Types

Caml is statically typed. The types are associated to both variables and values, and are checked for consistency at compile time. Moreover, the typing of expressions and variables can be performed automatically, without any explicit type declaration.

There are many possible definitions of what is meant by a data type. From the denotational perspective, a type is a set of values, called the *carrier set* of the type. Constructively, a type is viewed as a building process of its values. The abstraction-based viewpoint sees a type as a set of operations over a set of values, specifying explicitly the semantics of the operators. Here we prefer the last two perspectives. As far as the notation is concerned, **v:T** and **v∈T** mean the same thing: the type of the construct **v** is **T**.

As an algebraic abstraction, a type is a triple **T= <V,Op,Ax>** where **V** is the set of type values (the carrier set of the type), **Op** is the set of type operators (including the signatures of the operators), and **Ax** is a set of axioms that describe the behavior of the operators. This definition favors the constructive perspective of the values of the type. For example, using an ad-hoc notation, we can describe the type **list** with elements of type α as:

```
type α list is
Op
  []:→ α list                 ; list constructors
  :: :α × α list → α          ; [] is the empty list, :: is like cons

  hd: α list \ {[]} → α       ; list selectors
  tl: α list \ {[]} → α list ; hd (head) is as car, tl (tail) is as cdr

  null: α list -> bool        ; list predicates

Ax
  L: α list, x: α

  null [] = true              ; testing for the empty list
  null (x::L) = false

  hd(x :: L) = x              ; taking the head of a list
  tl(x :: L) = L              ; taking the tail of a list
```

Often, in a programming language the type is identified with its carrier set. However, the operators – in particular the constructors – are important if the language provides for both the cliché and the symbolic, constructive, representation of the values of a type. In Caml both variants are possible.

An interesting example is the definition of a function by points. After all, a function is a restricted relation over the Cartesian product of two sets that can be seen as the carrier sets of types. Consider the Caml definition of the **hd** operator as described in the algebraic presentation of the **list** type.

```
let hd = fun []     ->  raise (failure "hd on empty list")
           | (x::_) -> x;;
```

The definition is read: for the point **[]**∈**list**, the function **hd** is not defined; for a point **(x::_)**∈**list**, built by applying the constructor **::** on **x:**α and on an anonymous value **_:** α **list**, the value of the function is **x**. Here, the constructive view of values of type **list** is essential.

Another similar example is the appending of two lists. In the first variant a list is considered as an opaque value the parts of which are obtainable using the selectors **hd** and **tl**.

```
let rec append  A B =
    if null A then B else (hd A)::(append (tl A) B);;
append: 'a list -> 'a list -> 'a list = <fun>
```

In the second variant a list **L** is considered either as the empty list **[]** or as a composite value constructed by inserting an element **x** into another list **Ls** i.e. **L = x::Ls**.

```
let rec append = fun []  B -> B
                  |  (x::Ls) B -> x::(append Ls B);;
append: 'a list -> 'a list -> 'a list = <fun>
```

The definition above is based on pattern matching. In turn, the pattern matching is enabled by adhering to the constructive viewpoint of representing the values of a list. The first parameter of **append** matches one of the templates **[]** and **x::Ls**. As the result of the pattern match, performed sequentially, the variables **x** and **Ls** are bound to the top element of the list and to the tail of the list. Notice that this way of defining a function eliminates the need for eventual comparisons of the arguments against given values. This is quite important when the arguments are compound values that contain functions.

Constructing types

Each programming language comes with a universe of predefined types and types that can be built by the user. Call this universe of types **MT** and consider that it is a sort of meta-type whose values are types. In addition, a language offers type constructors, in fact functions $c:MT^n \rightarrow MT,\ n\geq 0$. In the particular case of Caml, the **0**-ary type constructors (i.e. predefined types) are: **int** (the integer type), **bool** (the Boolean type), **float** (the floating-point type), **char** (the character type), **string** (the string type), **unit** (the empty type), and **exn** (the exception type).

The conventional representation of the values of most of these types (i.e. the constants of these types) is as in Scheme. The operators are the same as in most common languages. Exception is the **bool** type, where the constants are **false** and **true**, and the operators are **&** (the boolean **and**), **or**, and **not**. As in the case of a conventional type, we can define variables over **MT**. Such a *type variable* can be bound to a value from **MT**, i.e. to a type. In Caml, type variables are represented as **'*identifier***.

Using type constructors, type variables and constants from **MT** (predefined types) we can build type expressions that represent new types from **MT**. For example, Caml offers the following type constructors:

- **`*`**$:MT^n \rightarrow MT$ The type expression $\alpha_1 * \alpha_2 * ... * \alpha_n$, $\alpha_i \in MT$, builds the type corresponding to the Cartesian product $\alpha_1 \times \alpha_2 \times ... \times \alpha_n$. A constant of the type $\alpha_1 * \alpha_2 * ... * \alpha_n$ is represented as a tuple $(a_1, a_2, ..., a_n)$, $a_i : \alpha_i$. For the particular type $\alpha_1 * \alpha_2$ there are two predefined operators: $fst(a_1, a_2) = a_1$ and $snd(a_1, a_2) = a_2$.

- **list**$:MT \rightarrow MT$ The type expression α **list**, $\alpha \in MT$, builds the type corresponding to lists with elements of type α. A constant of the type α **list** is represented as $[a_1; a_2; ...; a_n]$, $a_i : \alpha$. Apart from **null**, the common operators for the type α **list** are those from the algebraic presentation of the type α **list** above: **::** and **[]** (the empty list) are the list constructors, **hd** and **tl** are the list selectors.

Notice that the list operators work on any list α **list** regardless the value of α. They are *polymorphic*. In addition, the empty list is polymorphic. Polymorphism must be differentiated from *overloading*. In the case of overloaded names, these names correspond to different functions (eventually methods of a class in an OOP language) performing similar or different operations. In the case of general polymorphism there is single function performing the same operation on parameters with a generic type. In OOP a restricted form of polymorphism can be achieved by building taxonomies of classes. The derived classes inherit the methods of the parent class; hence, a method from a parent class can be used for any derived class.

- Interesting categories of type constructors are those that build *sum types*. These constructors are not predefined. They must be explicitly defined by using the expression

$$\textit{type id} = ncon_1 \mid ncon_2 \mid ... \mid ncon_m \mid$$
$$con_1 \text{ of } \alpha_1 \mid con_2 \text{ of } \alpha_2 \mid ... \mid con_n \text{ of } \alpha_n$$

The expression builds a sum type constructor called ***id*** whose symbolic values are either $ncon_j$ or $con_i(a_i)$, $a_i : \alpha_i$. Here $ncon_j$ and con_i are identifiers that play the role of constructors of the values of type ***id***, i.e. $ncon_j : \rightarrow id$ and $con_i : \alpha_i \rightarrow id$.. These constructors do not have an implementation; there is no code behind their names. They are symbolic. As an example, consider defining the type of natural numbers and inventing arithmetic operators that work with symbolic values of numbers. First, consider a possible algebraic specification of the type **natural**.

```
type natural is
Op
  zero: → natural                  ; basic constructors
  succ: natural → natural

  pred: natural\{zero} → natural   ; simple arithmetic operators
  add:  natural * natural → natural
  dif:  natural * natural → natural

  gt?: natural * natural → bool    ; predicates
  eq?: natural * natural → bool

Ax
  pred(succ(n)) = n

  add(n,zero) = n
  add(n,succ(m)) = succ(add(n,m))

  dif(n,zero) = n
  gt?(n,m) or eq?(n,m) => dif(succ(n),succ(m)) = dif(n,m)
```

```
  eq?(zero,zero)    = true
  eq?(succ(n),zero) = false
  eq?(zero,succ(n)) = false
  eq?(succ(n),succ(m)) = eq?(n,m)

  gt?(zero,zero)    = false
  gt?(succ(n),zero) = true
  gt?(zero,succ(n)) = false
  gt?(succ(n),succ(m)) = gt?(n,m)
```

The Caml representation of a **natural** number uses a sum type which explicates the basic constructors: **zero** and **succ**. The implementation is a rewriting of the axioms from the specification and adds some more code to cater with possible errors.

```
type natural = zero | succ of natural;;

exception natural_error of string;;
let nat_err = fun s -> raise (natural_error s);;

let rec
    pred = fun (succ n) -> n
            |  zero     -> nat_err "pred"
and add =  fun n zero     -> n
            |  n (succ m) -> succ(add n m)
and dif =  fun zero     (succ n) -> nat_err "dif"
            |  n        zero     -> n
            |  (succ n) (succ m) -> dif n m
and eq =   fun zero     zero     -> true
            |  (succ n) (succ m) -> eq n m
            |   _        _       -> false
and gt =   fun (succ n) zero     -> true
            |  (succ n) (succ m) -> gt n m
            |   _        _       -> false;;

(* Infix variants for add, dif, gt? and eq? *)
let prefix + x y = add x y
and prefix - x y = dif x y
and prefix > x y = gt x y
and prefix = x y = eq x y;;

(* Multiplication *)
let rec prefix * n m =
    if m = zero then zero else n + (n * (pred m));;

(* Some operations with symbolic numbers *)
let one = succ zero;;
let two = succ one;;
let three = succ two;;

two * (three + two);;
- : natural= succ(succ(succ(succ(succ(succ(succ(succ(succ(succ zero)))))))))

two * (three - one) = (succ three);;
- : bool = true
```

Sum type constructors can be parameterized. The **1**-ary sum type constructor **nested_list** below describes a list that can contain values of a generic type and, recursively, nested lists of the same kind.

```
type 'a nested_list = atom of 'a
                    | branch of 'a nested_list;;
```

Since it is a type constructor, **`nested_list`** can be applied on a type to obtain a specific nested list type. For instance, **`int nested_list`** is the type of nested lists that contain integers. The type **`('a nested_list) nested_list`** corresponds to polymorphic nested lists that contain nested lists with elements of type **`'a`**.

It is obvious that by combining the type constructors mentioned so far, complex types can be built. However, Caml is a functional language and since it is strong-typed we expect an additional, outstanding, type constructor.

- **`->:MT`** $\rightarrow$ **`MT`** The application α**`->`**β builds the type whose values are **`1`**-ary functions with the domain α and the range β. For example **`int->int`** is the type of all functions from **`int`** to **`int`**. If we consider **`->`** as right associative then **`int->int->int`** is the type of functions that return functions of the type **`int->int`**. A constant of the type α**`->`**β is written **`fun`** ***`fp`*** **`->`** ***`expression`***, where the type of the formal parameter is ***`fp`*****`:`**α, and the type of the ***`expression`*** (of the value computed) is β.

As far as the functions with no parameters are concerned, they are represented as **`1`**-ary functions the domain of which is the predefined type **`unit`**: a type with a single value, the empty tuple **`()`**. The function **`fun()->1`** is a function that when called with the empty tuple **`()`** returns always **`1`**.

The values of different types are built using the constructors specific to these types. Destroying a value and reclaiming the occupied space is performed automatically. As in Scheme, the values are garbage collected as long as they are not referenced in the program or from within a data structure used in the program.

3D Programming

In Scheme we noticed that programming is 2D, one dimension corresponds to the logic of the program, the other dimension is the evaluation time of the program expressions. Here, a third dimension is added: the types of expressions. When we build programs, we have to make sure the types of the expressions we are writing correspond to what we are thinking of. Taking into account the meaning of the expressions is crucial in programming. Meaning means typing. Thus thinking about types while programming can prove an important tool for building correct programs.

As an example, consider redefining the stream mechanism as implemented in Scheme. First we have to define the type *stream* of elements of a generic type **`'a`** (a type variable).

```
type 'a stream =   nil  |  term of 'a * (unit -> 'a stream);;
```

The symbolic value **`nil`** is the empty stream. The symbolic value **`term(`*****`t,function`*****`)`** represents a term of a non-empty stream: ***`t`*** is the value of the term, ***`function`*** is the closure able to generate the next term of the stream. In Scheme there was no restriction concerning the value returned by the ***`function`***. It took self-discipline to ensure that the ***`function`*** generates a value with the same meaning as the stream itself, i.e. a value that has the same type as the type of the stream. Here the type definition restricts subsequent processing of a term according to the described meaning.

Some useful stream operators are:

- **`(cons t f)`** builds the stream **`term(t,f)`**. It is more convenient in writing than **`term`**.
- **`hds(term(t,f))`** returns **`t`**, the first value of the stream.
- **`tls(term(t,f))`** returns the stream term generated by the application **`f()`**.

```
type 'a stream =   nil  |  term of 'a * (unit -> 'a stream);;
exception StreamError of string;;

let cons x f = term(x,f)
and hds = fun nil        -> raise(StreamError "hds on nil")
           | (term(x,_)) -> x

and tls = fun nil        -> raise(StreamError "tls on nil")
           | (term(_,f)) -> f();;

cons: 'a -> (unit -> 'a stream) -> 'a stream = <fun>
hds: 'a stream -> 'a = <fun>
tls: 'a stream -> 'a stream = <fun>
```

As an exercise in stream building, consider the stream of the terms of π:

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \ldots$$

In addition, let us define a function able to take the first **n** terms of a stream and return them as a list, and another able to generate the tail of a given stream by dropping the first **n** terms. Notice that equality comparison between the terms of a stream is avoided by using pattern matching, therefore by adhering to the constructive view of a value. Otherwise, we could run into trouble since the terms of a stream contain functions.

```
let pi =
    let rec pi_stream k =
        cons (4.0/.k)
             (fun () -> pi_stream (if k <. 0.0 then 2.0-.k else -.(k+.2.0)))
    in
        pi_stream 1.0;;
pi: float stream = term(4,<fun>)

let rec take n =
    fun nil -> []
     |  s   -> if n = 0 then []
                          else (hds s)::(take (pred n) (tls s));;
take: 'a stream -> int -> 'a list = <fun>

let rec drop n =
    fun nil -> nil
     |  s   -> if n = 0 then s
                          else drop (pred n) (tls s);;
drop: 'a stream -> int -> 'a stream = <fun>
```

Working with non terminating computations

The π series can be taken as base for an interesting computation. Note $\pi = t_1\ t_2 \ldots\ t_i \ldots$ the series of π and define π_i the approximation of π computed by summing the first **i** terms from the series, **i**>**0**. Consider building the stream

pi_approx = $\pi_1\ \pi_2\ \ldots\ \pi_i\ \ldots$

that contains successive approximations of π.

An interesting way of doing it is suggested by the computation ⌘ represented below. ⌘ adds an infinity of non-finite streams. The stream from line **j** is the stream from line **j-1** shifted one position to the right. The addition of streams is performed on columns by adding all the numbers in a column, as illustrated in figure 1.

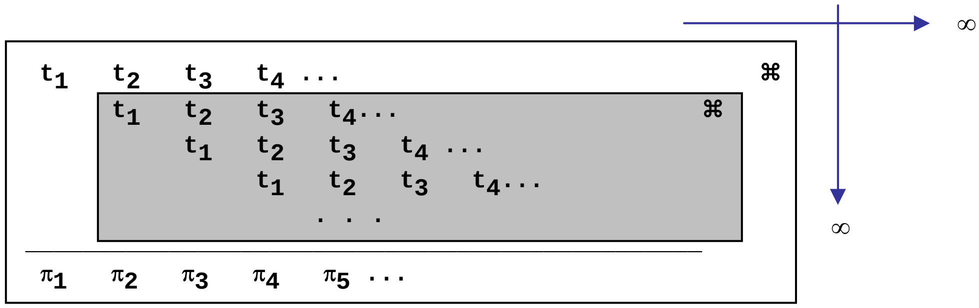

Figure 1 Computing the stream of successive approximations of π

The operation ⌘ can be described recursively as

$$\text{⌘}(\pi) = t_1\,(\text{tls}(\pi)\ \text{✦}\ \text{⌘}(\pi))$$

where the operation ✦ adds two streams term by term, column-wise. Written in Caml the story looks like below, where **add** corresponds to the ✦ operation and **tower_add** corresponds to ⌘.

```
#infix "add";;

let rec prefix add =
    fun nil b   -> b
     |  a   nil -> a
     |  a   b   -> cons ((hds a) +. (hds b))
                        (fun () -> (tls a) add (tls b));;
add: float stream -> float stream -> float stream = <fun>

let rec tower_add =
    fun nil -> nil
     |  s   -> cons (hds s)
                    (fun () -> (tls s) add (tower_add s));;
tower_add: float stream -> float stream = <fun>

let pi_approx = tower_add pi;;
pi_approx: float stream = term(4.,<fun>)
```

We managed to express the stream **pi_approx** as the result of a non-terminating operation performed on an infinite number of non-finite arguments. However, the computation of **pi_approx** terminates, provided we are interested in a finite number of successive approximations of π. We can now write a function that approximates π with a given precision and returns the number of terms of the π series that must be summed. The function can be easily generalized.

```
let approx epsilon =
    let rec
        abs x = if x <. 0.0 then 0.0 -. x else x
    and
        cycle s n =
              if abs(hds(s) -. hds(tls s)) <=. epsilon
                 then n,(hds s)
                 else cycle (tls s) (succ n)
    in cycle pi_approx 1;;
approx: float -> int*float = <fun>
```