Functional Programming with Static Typing in Haskell

Opposite to Scheme, 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 according to 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 Haskell. Almost all that has been discussed for Scheme applies to Haskell 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

```
\ formal params -> expression
```

and, as in Scheme, produces a functional closure. However, Haskell is a fully statically scoped language. Therefore, top-level variables are statically scoped and are visible throughout the entire program module that contains their definition. The top-level special expression:

```
variable = expression
```

is similar to the **define** top-level binding expression in Scheme. It binds the ***variable*** to the value of the ***expression***. 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!**.

```
plus = \ x -> (\ y -> x + y)
plus:: int -> int -> int

factorial = \ n -> if n==1 then 1 else n*factorial(n-1)
factorial:: int -> int
```

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 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 encountered in Scheme are ruled out. The variable **factorial** is saved within the functional closure bound to **factorial** and it is statically scoped.

The application of a function follows the normal-order evaluation: parameters are passed unevaluated. Moreover, the evaluation of a parameter is performed once only and its cached value is then used whenever required. The syntax of a 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 **\ *p1 p2 … pn -> expression***, similarly to the convention in the λ_0 language. The function **plus** can be written:

```
plus = \ x y -> x + y
plus:: int -> int -> int
```

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

```
plus x y = x + y
plus:: int -> int -> int
```

Apart from Scheme, the functions can be used both in infix or prefix form. For instance the application **(+) x y** is equivalent to the more familiar **x+y**, and **mod x y** can be written **x `mod` y**. Therefore **(op)** corresponds to the prefixed form of an infix binary operator, while **`op`** corresponds to the infixed form of a prefix binary operator.

```
comp f g = (\x -> f (g x))
comp:: (b -> c) -> (a -> b) -> a -> c

ff = (\ x -> x*x) `comp` (\ x -> x+x)
ff:: Integer -> Integer

ff 2
16
```

In the example above, the types in the signature of the composition function **comp** are not constants. They are generic types represented by type variables. The function **comp** is polymorphic. A function with the signature **sig** can take arguments of any type that obey **sig**.

As in Scheme, there are scoping expressions. The **let** from Scheme has similar equivalents in Haskell.

let {*var*$_1$ = *expr*$_1$; *var*$_2$ = *expr*$_2$; … ; *var*$_n$ = *expr*$_n$} in *expr*

***expr* where {*var*$_1$ = *expr*$_1$; *var*$_2$ = *expr*$_2$; … ; *var*$_n$ = *expr*$_n$}**

The scope of ***var*$_i$ i=1,n** is the entire expression that contains the definition of ***var*$_i$**. The variable ***var*$_i$** is bound to the unevaluated expression ***expr*$_i$** (the evaluation in Haskell is lazy). The result of ***expr*** is the result of the **let** expression.

```
is_even =
    let {is_even n = n == 0 || is_odd(n-1);
         is_odd  n = n /= 0 && is_even(n-1)}
    in is_even
is_even:: Integer -> Bool
```

```
is_even' = is_even where
    {is_even n = n == 0 || is_odd(n-1);
     is_odd  n = n /= 0 && is_even(n-1)}
```

The **||** operator is the Boolean or and **&&** is Boolean and. Both are 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**.

Indentation

The curly brackets **{** and **}** and the separator **;** can be omitted according to the following indentation rules:

- If the bracket **{** is missing after **where**, **let**, **of** and **do** then:
  - A new indentation paragraph is opened.
  - The bracket **{** is automatically inserted in the program text.
  - The starting position poz_0 of the next lexeme, say lex_0, is saved and becomes the alignment position of the current indentation paragraph.
- If the lexeme lex_i, i>0, starts at position $poz_i > poz_0$ then lex_i is taken as the continuation of lex_{i-1}
- If the lexeme lex_i, i>0, starts at position $poz_i = poz_0$ then the separator **;** is automatically inserted after the lexeme lex_{i-1}.
- If the lexeme lex_i starts at position $poz_i < poz_0$:
  - A curly bracket **}** is automatically inserted for each indentation paragraph that has the indentation position greater than poz_i (hence for the current paragraph as well).

For example, the **is_even** and **is_even'** functions can be written as shown below, omitting the curly brackets and the **;** separator.

```
is_even =
    let is_even n = n == 0 || is_odd(n-1)
        is_odd  n = n /= 0 && is_even(n-1)
    in is_even

is_even' = is_even where
        is_even n = n == 0 || is_odd(n-1)
        is_odd  n = n /= 0 && is_even(n-1)
```

Types

Haskell 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

  head: α list \ {[]} → α      // list selectors
  tail: 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

  head(x : L) = x              // taking the head of a list
  tail(x : L) = L              // taking the tail of a list
```

Often, in a programming language a type identifier is the identifier of the type carrier set. 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 Haskell 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 definition of the **head** operator as described in the algebraic presentation of the **list** type.

```
head (x:_) = x
```

The definition is read: for a point **(x::_)∈ α 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 **α 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 **head** and **tail**.

```
append a b =
    if a==[] then b else (head a):(append (tail a) b)
append:: [a] -> [a] -> [a]
```

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**.

```
append []  b = b
append (x::ls) b = x:(append ls b)
append:: [a] -> [a] -> [a]
```

The above definition 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.

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 $\mathbf{c:MT^n \rightarrow MT,\ n \geq 0}$. In the particular case of Haskelll, the **0**-ary type constructors (i.e. predefined types) are: **Integer** (the integer type), **Bool** (the Boolean type), **Float** (the floating-point type), **Char** (the character type), **String** (the string type), **()** (the empty type, called *unit*), etc[1].

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 Haskell, a type variable is 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, Haskell offers the following type constructors:

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

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

Notice that the list operators work on any list $\mathbf{[\alpha]}$ 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 parametric polymorphism (as in Haskell) 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 hierarchies 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

$$\mathbf{data}\ id\ v_1\ ...\ v_m = ncon_1 \mid ncon_2 \mid . \ . \ . \mid ncon_m \mid con_1\ \alpha_1 \mid con_2\ \alpha_2 \mid . \ . \ . \mid con_n\ \alpha_n$$

where $\mathbf{m \geq 0}$, v_j is a type variable, and α_i is a type expression parameterized on v_j, $\mathbf{j=1..m}$..

[1] The Haskell representation of the constants of most of these types and the associated operators is common to most programming languages. For example, the **Bool** constants are **False** and **True**, and the operators are **&&** (the boolean **and**), **||** (the boolean **or**), and **not**.

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 purely 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 Haskell 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.

```
data Natural = Zero | Succ Natural deriving Show[2]

pred (Succ n) = n
add n Zero = n
add n (Succ m) = Succ(add n m)

dif n Zero = n
dif (Succ n) (Succ m) = dif n m

eq Zero Zero = True
eq (Succ n) (Succ m) = eq n m
eq _ _ = False
```

[2] The names of types and of value constructors must start with a capital letter. The additional specification **deriving Show** makes it possible to display the symbolic values of **Natural**. The complete explanation is quite extensive and relates to the notion of **class** as viewed in Haskell.

```
gt (Succ n) Zero = True
gt (Succ n) (Succ m) = gt n m
gt _ _ = False

-- Multiplication
times n m = if eq m Zero then Zero else n `add` (times n (pred m))

-- Some operations with symbolic numbers
one = Succ Zero
two = Succ one
three = Succ two


two `times` (three `add` two)
Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ Zero))))))))):: Natural

(two `times` (three `dif` one)) `eq` (Succ three)
True::Bool
```

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.

```
data Nested_list a = Atom a
                   | Branch (Nested_list a) deriving Show
```

Since it is a type constructor, **Nested_list** can be applied on a type to obtain a specific nested list type. For instance, **Nested_list Integer** is the type of nested lists that contain integers. The type **Nested_list (Nested_list a)** 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, Haskell is a functional language and since it is strong-typed we expect an additional, outstanding, type constructor.

- **->:MT → MT** The application **α->β** builds the type whose values are **1**-ary functions with the domain **α** and the range **β**. For example **Integer->Integer** is the type of all functions from **Integer** to **Integer**. If we consider **->** as right associative then **Integer->Integer->Integer** is the type of functions that return functions of the type **Integer->Integer**. A constant of the type **α->β** is written **\ *fp* -> *expression***, where the type of the formal parameter ***fp*** is **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 **()** – called the unit type –: a type with a single value, the empty tuple **()**. The function **\()->1** is a function that when called with the empty tuple **()** returns always **1**.

The values of different types are built using constructor operators 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 by a program variable or from within a data structure used in the program.

Working with non terminating computations

Consider computing with streams, as we did in Scheme. First, observe that streams are quite natural in Haskell: a list is in fact a stream (Haskell is using lazy evaluation). Therefore, there

is no need for the sort of complications we've been through when implementing streams in Scheme. We can work directly with lists.

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.

```
pi = pi_stream 1.0 where
        pi_stream k = (4.0/k):(pi_stream (if k < 0.0 then 2.0-k else -(k+2.0)))
pi::[Double]

take :: Integer -> [a] -> [a]  -- Explicit declaration of signature
take n [] = []
take 0 _ = []
take (n+1) (x:xs) = x : (take n xs)

drop :: Integer -> [a] -> [a]
drop n [] = []
drop 0 xs = xs
drop (n+1) (x:xs) = drop n xs
```

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.

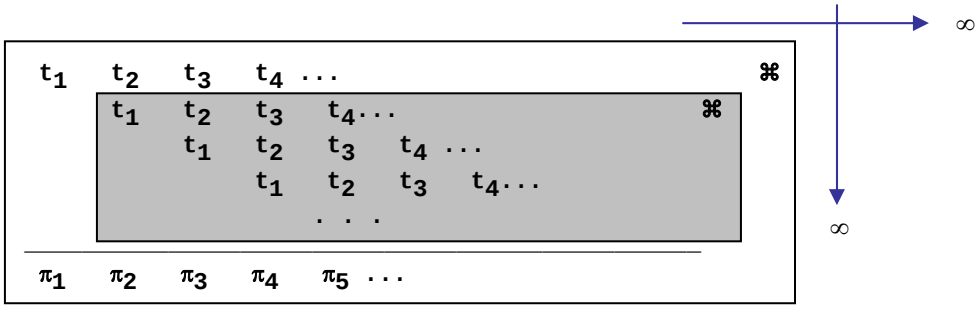

Figure 1 Computing the stream of successive approximations of π

The operation ⌘ can be described recursively as

$$⌘(\pi) = t_1\ (tls(\pi) \ ✦\ ⌘(\pi))$$

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 ⌘.

```
add :: [Double] -> [Double] -> [Double]
add [] b = b
add a [] = a
add (x:a) (y:b) = (x+y):(add a b)

tower_add [] = []
tower_add s = (head s):((tail s) `add` (tower_add s))
tower_add:: [Double] -> [Double]

pi_approx = tower_add pi
pi_approx:: [Double]
```

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.

```
approx :: Double -> (Integer,Double)
approx epsilon =  cycle pi_approx 1
   where
        abs x = if x < 0.0 then 0.0 - x else x
        cycle s n =
              if abs(head(s) - head(tail s)) <= epsilon
                 then (n,(head s))
                 else cycle (tail s) (n+1)

approx 0.0002
(10000,3.14149265359003) :: (Integer,Double)
```