Bordeaux University
Year 2015-2016

Master ESSV
MtoCPL

TD n°4 - Constrained polymorphism

▷ Haskell is a purely-functional programming language with cutting-edge type-checking abilities. The compiler used here is the Glasgow Haskell Compiler, accessible with the command `ghc`, and its interaction loop `ghci`.

In order to write a Haskell program, you can open a `.hs` file in `emacs`, and compile it with the following command :

```
ghc "file.hs"
```

Alternatively, it is possible to start the interaction loop and load the file.

```
:load file.hs
```

Notice that `ghci` provides auto-completion for a lot of things.

The translation between λ-calculus, OCaml and Haskell code goes like this :

	λ-calculus	OCaml	Haskell
Conditional	`if then else`	`if then else`	`if then else`
Application	`t u`	`t u`	`t u`
Abstraction	`λx.t`	`fun x → t`	`\x → t`
	`λx:T.t`	`fun (x:T) → t`	`\(x::T) → t` [1]
Multiple abstraction	`λx.λy.t`	`fun x y → t`	`\x y → t`

The types used for naturals will be `int` in OCaml (resp. `Int` in Haskell). The types used for booleans will be `bool` (resp. `Bool`). The rules of associativity for expressions are :

Expressions	Application is associative to the left	$fgh \equiv (fg)h$
Types	Arrow is associative to the right	$a \to b \to c \equiv a \to (b \to c)$

Both programming languages have the following properties :

- Displaying types : in OCaml, every expression `e` entered in the interaction loop is evaluated and its return value is displayed with its type. In Haskell, `:t e` displays the type of the expression `e`.

[1] The ScopedTypeVariables extension must be enabled in order to allow this construct.

- Naming values : in each language, it is possible to give a name to a value at the toplevel using the following construct :

OCaml	Haskell
`let ⟨name⟩ = ⟨expr⟩`	`⟨name⟩ = ⟨expr⟩`

- Syntaxic sugar for functions : instead of writing a series of `fun`, it is possible to define a function with multiple arguments using the following construct :

`let ⟨f⟩ ⟨arg1⟩ ⟨arg2⟩ = ⟨e⟩` $\equiv$ `let ⟨f⟩ = fun ⟨arg1⟩ → fun ⟨arg2⟩ →⟨e⟩`

Exercice 1: Haskell polymorphism

In the Haskell interaction loop, it is possible to retrieve the type of a value :

```
let f = \x → x
:t f  −− ⇒ f :: t → t
```

The following functions allow to use our language in Haskell :

```
succ :: Int → Int
succ x      = x+1
pred :: Int → Int
pred x      = if (x<=0) then 0 else (x−1)
is_zero :: Int → Bool
is_zero x  = (x == 0)
empty      = []           −− the empty list
cons x e   = (x:e)        −− cons of a head and a tail
is_empty l = (l == [])    −− tests if a list is empty or not
```

For each of the following functions, first compute their generic type and then verify it by writing the corresponding Haskell code :

1. the projection function, that takes n parameters and returns the k-th (take for example $n = 5$ and $k = 3$).

2. a fixpoint-search function, that takes a function f and a starting point x_0, and returns (when terminating) a fixed point of f (the following function can be used for testing : `\x → if (x <= 5) then x + 1 else x`).

   What is the meaning of `Eq a` in the type of this function ?

Haskell has a type called `Maybe a` and two constructors for values in this type : `Just :: a → Maybe a`, and `Nothing :: Maybe a`. Concretely, a value of type `Maybe a` is either unknown (`Nothing`) or known with a value x (`Just x`). This type handles the cases where no return value is known for a function. To test if a value of type `Maybe a` is a `Nothing` or a `Just`, it suffices to use the `case` construct :

```
fromMaybe :: a → Maybe a → a        -- Take a default value and and Maybe value.
fromMaybe dflt may  = case may of -- If the Maybe is Nothing, return the default
  | Just val → val                  -- Otherwise, return the value in the Just
  | Nothing  → dflt
```

3. Write a `search` function within a list that returns the index of an element in a list and returns `Nothing` if not found.

Exercice 2: Interlude : Interfaces in Java

In the Java standard library, one can find generic interfaces over the type `T` (for example `Comparable<T>`, cf. https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) that define a set of generic functions over `T` (in our case `int T.compareTo(T o)`). Naturally, every class implementing this interface must also implement this set of functions.

Now it is possible to define code that depends on **all** the classes implementing such an interface.

1. Define a generic interface `Sortable<T>` that contains a `sort` function, such that the type variable `T` can only range over all classes implementing `Comparable<T>`.

2. Write a generic class `SortableLinkedList<T>` that extents `LinkedList<T>` and implements `Sortable<T>`. In order to implement the sort function, you can use `Collections.sort`.

```
Collections.sort(ls, new Comparator<?!>(){ // Replace ?! with correct type
       public int compare(?! o1, ?! o2) {
           return -1;
        }});
```

Exercice 3: Basic type classes

Type classes are Haskell constructs that, according to the definition of the *Real World Haskell* book, “define generic interfaces providing a common feature set over a wide variety of types”. They define a sort of constrained polymorphism. The simplest type class in Haskell is the `Show` class, which implements the `show` method as :

```
class Show a where
  show                    :: a → String
```

It is heavily used in the interaction loop, because the `show` function is used to print every value. The equivalent in Java would be the overloaded method `toString`. For example, when defining a new datatype for a polymorphic list `List a`,

```
data List a = Nil | Cons a (List a)
```

... it is impossible to even have a look at the value `Nil` in the interaction loop before implementing the `Show` typeclass for `List a`, by the creation of an **`instance`** :

```
instance Show a ⇒ Show (List a) where
         show Nil         = "[]"
         show (Cons x l) = (show x) ++ ";" ++ (show l)
```

Notice that in order to be able to "show" a list of elements of type `a`, it is necessary to be able to show an element of type `a`. This requirement is expressed with the expression `Show a ⇒ Show (List a)`. In the end, the type of the `show` function is :

```
show :: (Show a) ⇒ a → String -- :t show
```

1. As a warm-up, implement a `show` function that correctly displays a `List a`, for example `[]` if empty, and `[1;2;3;4]` when not empty.
2. Explore and explain the type classes representing the numeric types in Haskell. In particular, explain the types of `(/)`, `(%)`, and `(^)` (you can refer to `http://www.haskell.org/tutorial/numbers.html`).
3. Consider the classes `Eq a` and `Ord a` (for example in `http://www.haskell.org/tutorial/classes.html`). What functions do they implement ? What instances do they possess ?

Exercice 4: Haskell and JSON

Suppose that we want to represent JSON[2] data in Haskell. The type used to represent JSON values could look like [3] :

```
data JValue = JChar Char
            | JNumber Double
            | JArray [JValue]
              deriving (Eq, Ord, Show)

example1 :: [Double]
example1 = [1,2,3]

main = do
  putStrLn $ show $ JArray (map JNumber example1)
```

[2] Javascript Object Notation, a serialization format widely used in client-server exchanges.

[3] Example taken from `http://book.realworldhaskell.org/read/using-typeclasses.html`

Notice that this encoding allows recursive values. Moreover, thanks to the `deriving` expression, this datatype already implements the fonctions `(==)`, `(<)` and `show`. Now, it should be possible to transform usual values into JSON values and back. For this, we propose to implement the following type class :

```
class JSON a where
    toJValue   :: a → JValue
    fromJValue :: JValue → Maybe a

instance JSON JValue where
    toJValue   = id
    fromJValue = Just
```

1. Write an instance of the JSON typeclass for the type `Double`.
2. Write another instance for `[a]` where `a` is a type variable.

   Hint : the function `fromJValue` needs to propagate the first error encountered.
3. Why is this a form of constrained polymorphism ?
4. Inside a Haskell expression containing a call to the `fromJValue` function, how is the parameter `a` determined ?

Suppose now that we want to extend the behavior of our `JValue` with an unsafe method `unsafefromJValue` :

```
unsafefromJValue :: JValue → a
```

5. Write a new type class `JSON_Ext` that extends `JSON`, and the implementations corresponding to `JSON_Ext Double` and `JSON_Ext [a]`.
6. What is the behavior of the unsafe version when no correct solution exists ? For example, how do you explain the behavior of the following calls :

```
example2 :: JValue
example2 = JArray ([JNumber 2.0, JNumber 3.0])

example3 :: JValue
example3 = JArray ([ JChar 'a', JNumber 3.0])

(unsafefromJValue example2)::([Double])
(fromJValue example3)      ::(Maybe [Double])
(unsafefromJValue example3)::([Double])
```