Tutorial #4 - Parametric polymorphism

Exercice 1: Calculus on types

Metaprogramming is a programming technique that consists in generating and/or modifying programs with the help of other (meta-)programs. The C++ language allows metaprogramming through the use of *templates*. Within this system, it becomes possible to perform computations and code generation at compile-time.

Here, we define the booleans and the natural integers as compile-time values :

```
// Booleans
struct True  { static const bool value = true;  };
struct False { static const bool value = false; };

// Naturals
struct Zero  { static const int value = 0; };
template <typename N>
struct Succ  { static const int value = N::value + 1; };
```

1. Compile the examples in the source and check that `Succ` can be used as a compile-time function.

Instead of storing values inside structs, it is possible to store types. The following examples shows how to store a type inside a `struct` :

```
template <typename T>
struct Box {                // Box is a function
  typedef T res;            // of the type 'T'
};

Box<int>::res  ibx = 1;     // Apply Box<T>
Box<char>::res cbx = 'c'; // to T='int'/'char'
```

```
template <typename T>
struct Copy {     // Copy<T> uses typename
  typedef typename T::res res;
};

Copy<Box<char>>::res ic = 'y';
```

From now on, we propose to encode the simply-typed λ-calculus with C++ metaprogrammation. However, all the computations will be done *exclusively on types*.

2. What is, in this context, the equivalent of a type variable ? An application ? An abstraction ?

   In order to answer this question, write the `fst` and `snd` functions that take two parameters and return respectively the first and the second.

3. Write the equivalent of an `if-then-else` construct.

4. Write the equivalent of an `equal` function.

These examples show that it is possible to perform some computations at compile-time on types. At best, these computations should be the same as those possible in the λ-calculus. What is missing in order to get that kind of expressivity ?

5. What is the equivalent of *recursion* within this framework ?
6. To what extent is this implementation of the simply-typed λ-calculus within the types of C++ close from the 2^{nd}-order polymorphic calculus ?

Exercice 2: Constrained genericity

In OCaml, the type of the comparison operators is very generic :

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

This means that the comparison operators may be applied to any pair of values, as long as they possess the same type. This is problematic because there are values that are difficult, or even impossible to compare.

1. Invent two values that are impossible to compare and try to compare them in OCaml (*Hint* : these values are present in the simply-typed λ-calculus).

▷ 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`.

The executables are available in the `/net/ens/renault/haskell/bin` directory.

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 `ghci` and load the file.

```
:load file.hs
```

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

In Haskell, the comparison operators have a different type :

```
:t (==) -- → (==) :: Eq a ⇒ a → a → Bool
```

This type is the example of a *type class*. A type class such as `Eq a` can be seen as a set of types possessing a property dependent on the variable `a`. In the case of type classes, the property is very similar to a generic interface that the type must respond to. For `Eq a`, the interface is the following :

```haskell
class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool
```

In order to define a type class for a specific type, it is just necessary to provide a specific implementation for the functions in the type class for the particular types. In the following example, we declare a new type and the associated `Eq` instance :

```haskell
data Sign = Plus | Minus

instance Eq Sign where
  (==) Plus  Plus  = True
  (==) Minus Minus = True
  (==) Plus  Minus = False
  (==) Minus Plus  = False
```

2. Find the types that belong to the `Eq a` type class.

   Note : although `https://www.haskell.org/hoogle` is a nice place to look for Haskell functions, for this very question, you may be better off with your usual search engine.

3. What would be the equivalent of the `Eq a` class in Java ?

Type classes can be related by inclusion. For example, the following definition derives an instance of `Eq` for lists of objects, provided a definition of `Eq` for these very objects.

```haskell
instance Eq a ⇒ Eq [a]
```

On the same note, they provide a very powerful manner to handle numeric types in a polymorphic manner. They are extensively described at `https://www.haskell.org/tutorial/numbers.html`.

4. What is the type of the addition function in Haskell ?

   Find examples of values that you cannot add in Haskell.

5. Draw the inclusion relations between all the numeric type classes in Haskell as an Euler diagram.

The implementation of type classes is described in the article *Implementing type classes* from J. Peterson and M. Jones. This article is available at the following :

`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.8206`

Reading this article, find out the mechanisms that allow the following code to type-check. In particular, write the code that is added by the compiler and the dictionary of the instances of the `Equal a` class.

```
class Equal a where
  equal :: a → a → Bool

instance Equal Int where
  equal x y = (x == y)
instance Equal Char where
  equal x y = (x == y)

instance Equal a ⇒ Equal [a] where
  equal x y = eq_list x y

eq_list []     []     = True
eq_list (x:xs) (y:ys) = (equal x y) && (eq_list xs ys)
eq_list _      _      = False

test = equal ['a','b','c'] ['d','e','f']
```