Tutorial #3 - Type inference

Exercice 1: Undefined behaviors

An interesting aspect of the theory of computation is the existence of *partial functions*. These functions introduce undefined behaviors in the evaluation function of a language. Simple examples are the predecessor function on naturals (`pred zero`) and the division on integers (`1/0`).

Another very general example of undefined behavior comes from the distinction between *initialized* and *uninitialized* values. For instance, in an object-oriented language, references to objects shall usually be allocated in a specific manner before being ready for computation. Usually, an uninitialized elements takes a particular value that depends on the language :

`None` in Python, `nil` in Lisp or Ruby, `null` in Java or C#, `nullptr` in C++ 11, ...

1. In a statically typed language, what type would you give to such a value ?
2. Select one of these statically typed languages and trick it to addition two references on numbers that are not initialized.
   What limit of type systems does this highlight ?
3. Propose a type for this kind of value (possibly a family of types) ? What kinds of solutions could you propose to alleviate this problem ?

Remark : Tony Hoare, one of the designers the language ALGOL W that introduced a `NULL` reference, has an interesting opinion on the subject (cf. `https://en.wikipedia.org/wiki/Tony_Hoare`).

Exercice 2: Type inference in OCaml

The OCaml compiler is able to infer the types of every expression. Its type system is an extension of the type system discussed in the course. It can effectively type functions, lists, arrays, records, algebraic data types, references, and even objects in a pretty powerful manner.

1. Try the type inference on the following values :

```
[1; 2; 3];;                    (* a list *)
[|'a'; 'b'; 'c'|];;            (* an array *)
(5, "a", true);;               (* a tuple *)
object method id = "x" end;;   (* an object *)
fun x y → x+y;;                (* a function *)
```

For each value, note the associated type and derive the form of a generic type representing all the values of this form.

2. What would be the type of a function transforming a list into an array ?
   In this context, what is the meaning of the type variable ?

References in OCaml are introduced via the `ref` keyword. For instance, `ref 1` is a reference containing the value `1`. Construction, assignment and extraction of the contents are done in the following manner :

```
let pi = ref 1;;
pi := 2;;
!pi + !pi;; (* → 4 *)
```

3. What is the type of `pi`, a reference on an integer ? Is it in some manner possible that this type be modified through the use of the reference ?
4. What is the type of a reference on an empty list ?
   Is it in some manner possiblt that this type be modified through the use of the reference ? What kind of property on types does this behavior illustrate ?
5. Try to type the following code :

```
let f0 = fun x → (x,x) in
let f1 = fun y → f0 (f0 y) in
let f2 = fun y → f1 (f1 y) in
let f3 = fun y → f2 (f2 y) in
let f4 = fun y → f3 (f3 y) in
let f5 = fun y → f4 (f4 y) in
  f5 (fun z → z)
```

What can you deduce about the type inference algorithm ?

Exercice 3: Type inference and programming

Consider the following classical implementation of the `map` function in OCaml :

```
let rec map f l = match l with
  | []    → []
  | x::xs → f(x)::(map f xs);;
```

This implementation uses the pattern-matching for the lists, where `[]` represents the empty list, and `x::xs` represents a list with head `x` and tail `xs`.

1. Give the types of the following sub-expressions : `l`, `f`, `(::)`, `map`.
2. Write a function with the same type and behavior in the Java programming language (*Challenge :* use the least number of type annotations possible, the use of version of Java superior to 10 is allowed)
3. How are the types related between the two languages ?
   What are the differences in terms of programming ?

Exercice 4: Type schemes

Consider the following expression in OCaml :

```
let f x = x in (f 1, f true);;
```

1. What are the constraints associated to the function `f` alone (meaning without the `in` part) and inside the whole expression ?
2. Are these constraints solvable with the type inference algorithm seen in class ?
3. Propose an adequate type for `f` and the whole expression.
4. Propose an algorithm for solving the constraints