Type Inference

Type systems come in many shapes and forms, each with its advantages and disadvantages. For instance the type systems in C++ and Java put a lot of emphasis on the idea of subtypes, while OCAML does not even allow subtyping.

OCAML's type system is what is known as the Damas–Hindley–Milner type system[1]. It has the remarkable property of being able to do type inference at linear time. we will discuss this feature in this section.

With type systems there are three fundamental types of questions we try to address:

1. Given an expression and a type, can we determine if the expression has that type? This is know as *type checking*.
2. Given an expression, can we determine its type, or *a* type for it? This is known as *type reconstruction* or *type inference*.
3. Given a type, can we produce a value that has that type? We will not consider this question much, but it is of some interest for type theorists.

The question of type inference is most pertinent for functions. Consider for example the following function:

```
let rec f lst =
    match lst with
    | x :: rest -> Some x :: f rest
    | [] -> [Some 1]
```

This expression contains enough information to work out the types of everything. Let us see how.

1. First of all, the system can immediately determine that f is a function, just by the syntax. Therefore the type of f must be t1 -> t2 where t1 and t2 are two types that we need to determine. We further know that the type of lst must be t1.
2. We then look into the body of the function. We find a match expression. Let us start with the first pattern. The interpreter notices the pattern recognizes a list, so it must be of some type t3 list, where x is of type t3 and rest is of type t3 list when we consider the right hand side. since lst is expected to match this pattern, we get our first "type equation": t1 = t3 list.
3. The right hand side of the first pattern is evaluated in an environment where rest has type t3 list and x has type t3. The interpreter sees that this must be a list, so it must have a type t4 list where t4 is the type of Some x and t4 list is the type of f rest.
    - Looking at the first of those two we deduce the equation t4 = t3 option.
    - The second has to match the type t4 list. But the second is a function call of f rest. We know that f has type t1 -> t2, so for the function call to even make sense we must have t1 = t3 list, and also the return type must match what we assumed about our list, so t2 = t4 list.

[1] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system

4. The right hand side of the second pattern suggests the equation t2 = int option list.

We therefore get a set of type equations:

```
t1 = t3 list
t4 = t3 option
t1 = t3 list
t2 = t4 list
t2 = int option list
```

Any expression results in a series of type equations like the above, with some unknown types to be determined. The process of solving these equations is called **unification**, and the standard algorithm for doing so is called Algorithm W.

Essentially this algorithm goes through each type equation in order, and depending on the equation takes an appropriate action. Along the way it keeps track of what it has learned. Let us see how:

The first equation for example tells is that it should associate the type t1 with t3 list. It will remember that and will replace any t1s present in future equations with t3 list. So our equations look like:

```
Remember:  t1 ---> t3 list
Remaining:
t4 = t3 option
t3 list = t3 list
t2 = t4 list
t2 = int option list
```

We then proceed to the second equation, from which we learn that we must associate t4 with t3 option. So now our state and equations become:

```
Remember:  t1 ---> t3 list
           t4 ---> t3 option
Remaining:
t3 list = t3 list
t2 = t3 option list
t2 = int option list
```

Next it will look at the equation t3 list = t3 list. It will notice that they are both lists and it will then equate the container types t3=t3. It then sets that equation.

```
Remember:  t1 ---> t3 list
           t4 ---> t3 option
Remaining:
t3 = t3
t2 = t3 option list
t2 = int option list
```

Now it reads t3 = t3, and that is a tautology without any information, so it discards the equation. We now have:

```
Remember:  t1 ---> t3 list
           t4 ---> t3 option
Remaining:
t2 = t3 option list
t2 = int option list
```

The next equation tells us to associate t2 with t3 option list. We will remember that and also substitute the t2 in the remaining equations:

```
Remember:  t1 ---> t3 list
           t4 ---> t3 option
           t2 ---> t3 option list
Remaining:
t3 option list = int option list
```

The last equation equates two lists, so it will instead equate their containers, so t3 option = int option goes into the list of equations. On the next step, this equation will be looked at and become t3 = int instead. Finally, that equation will be read and added to the list to remember, providing us with our final correspondence:

```
    t1 ---> t3 list
    t4 ---> t3 option
    t2 ---> t3 option list
    t3 ---> int
```

This is an example of the kind of work that algorithm W performs. The equations are meant to be read from the bottom up, and we could in fact go ahead and substitute each equation in all the others above it, to get:

```
    t1 ---> int list
    t4 ---> int option
    t2 ---> int option list
    t3 ---> int
```

So the algorithm determines that the function f must have type int list -> int option list.

The algorithm will also detect typechecking errors along the way. If you think about it for a second, this explains the somewhat obscure messages that the type-checker gives you: The errors some times are detected somewhat far from where they occur: Your first pattern might have something that will not cause problems until it is compared with the 4th pattern 10 lines later. The system has no real way of knowing which of the two had the error. Or sometimes the error might produce a valid function of the wrong type, which will only appear when you later call the function from another function, in perhaps an entirely different file. We will find some ways of limiting these kinds of problems.

Practice Problems

Work out the above steps, with the same level of detail, for the following examples:

```
let sum lst =
    match lst with
    | x :: rest -> x + sum rest
    | [] -> 0

let evp1 (f, x) =
    f (x + 1) + 1
```

```
let rec maybeAdd lst =
    match lst with
    | [] -> None
    | x :: rest -> match maybeAdd rest with
                   | None -> Some x
                   | Some y -> Some (x + y)
```