CIS 425 - Introduction to type inference

Functions in input

A *Higher-Order function* is a function which accepts other functions as arguments or returns a function as a result.

In ML, functions are *first-class citizens*, meaning that they have all the same operations available to them as other entities in the language do, including being passed as an argument to another function, and being returned from another function.

Example 1. Consider the following ML function definition:

```
fun apply (f, x) = f x;
```

What is the type of *apply*? We see that our function takes a tuple of two arguments, f and x. We see that we are applying the value f to x, so f must be a function. Since we do not know what the type of x is, nor what type f returns, we define the type of f as:

```
f : 'a -> 'b
```

Logically, x must be of type 'a, since that is what f takes in as arguments. Furthermore, we now have defined that f returns values of type 'b, so f applied to x must be a value of type 'b. This helps us to write the full type of *apply* as follows:

```
val apply = fn : ('a -> 'b) * 'a -> 'b
```

So, we see that the function *apply* is considered a higher-order function, because it is able to accept another function as an argument.

`apply` is a *polymorphic* function. For example, it can be invoked as follows: Example 1 above includes the use of what is known as a *polymorphic* function. A polymorphic function is a function which can evaluate to and/or be applied to values of different types.

Example 2. calling the apply function

Consider the same ML function, adding a call to apply:

```
fun apply (f, x) = f x;
apply (fn x => x+1, 5);
```

Note first that here, the function provided as the first argument to apply is an example of an *anonymous* function. *anonymous function* is exactly what it sounds like: a function without a name. Sometimes, all we need a function for is to pass it as an argument to another function, and so we can skip the need to define and name that function by using the **fn** expression.

In this example, the type of the anonymous function would in fact be

```
fn (x => x+1): int -> int
```

So, while in **Example 1** we saw that we had:

```
f : 'a -> 'b,
```

It is possible that 'a and 'b are the same, as they are in **Example 2**, where 'a : int and 'b : int, but it is not **required**.

Example 3. What if we changed our call to *a*pply to be the following?

```
apply (fn x => x + 1, true);
```

This would lead to an error! Why?

Because in our call to *a*pply where we give the anonymous function:

```
fn x => x + 1
```

we are mandating that the type of f be

```
f : int -> int
```

but then instead providing f with an argument of type bool.

Example 4. Consider several successive calls to *apply* as follows:

```
fun apply (f, x) = f x;
apply (fn x => x+1, 5);
apply (fn x => true, false);
apply (fn x => x + 1.0, 5.2);
```

Will this result in an error? I.e, does the first call to apply permanently set the required type of f? **No! And that is the beauty of polymorphic instantiation.** All of the above calls are permitted, even one after the other.

Functions in output

Since in ML, functions are considered first-class citizens, not only can we provide a function to another function as input, but we can also **receive** functions as output!

Example 5. Let us consider a new function, called apply' (read: *apply prime*), which we will then call with specific arguments.

```
fun apply' f = fn x => f x;
```

What is the type of apply'? Well, we first see that apply' takes in an argument **f**. We also see that f is being applied to x, so f itself must be a function. We also see that the return of apply' is actually an anonymous function, and that this anonymous function accepts an argument x, and then applies f to x. This give us the type of apply' as follows:

```
val apply' = fn : ('a -> 'b) -> 'a -> 'b
```

Say we called apply' as follows:

```
val a =   apply' (fn x => x + 1);
```

What would this return? it will return a function:

```
val  a = fn : int -> int
```

which can then be invoked as so:

```
a 5
```

which returns **6.**

Currying

Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each takes a single argument.

Currying is what allows us to perform something called *partial application*, wherein not all the parameters to a curried function need to be provided all at once.

Example 6. For instance, consider the following function definition:

```
fun f x1 x2 x3 = e;
```

This is an example of a curried function, where the arguments `x1`, `x2` and `x3` do NOT need to be supplied all at the same time. To better understand this, consider a second function definition:

```
fun f' x1 = fn x2 => fn x3 => e;
```

The function `f` is simply what we call *syntactic sugar* for f'. *Syntactic sugar* is syntax within a programming language that is designed to make things easier to read or to express. While `f` and `f'` appear syntactically different, they are functionally equivalent.

As a reminder, application associates to the left:

e1 e2 e3

is to be interpreted as:

((e1 e2) e3)

Example 7. Consider another series of function definitions:

```
fun sum (x, y, z) = x + y + z;
fun add x y z = x + y + z;
fun add' x => fn y => fn z => x + y + z;
```

As per Example 6, *add'* is simply syntactic sugar for *add*. But are sum and add equivalent? No, and to see why, we will look at their types and how we are able to call them. The type of *sum* is:

```
val sum = fn : int * int * int -> int
```

What this means is that *sum* accepts three ints in a tuple as its argument, and returns a single int. On the other hand, the type of *add* is:

```
val add = fn: int -> int -> int -> int
```

What does this mean? This means that, unlike *sum*, *add* does NOT need all of the three integers it accepts to compute its addition to be given all at once. In fact, trying to do something like:

```
add (1, 2, 3);
```

Will result in an error.
Why? because you are supplying the function with a tuple of type (int * int * int), when we only asked for an int!

Example 8. So how can we call add? In any of the following ways:

```
fun add x = fn y => fn z => x + y + z;

add 1;
add 1 2;
add 1 2 3;
```

Remember again that application is left to right: add 1 2 3 is to be interpreted as (((add 1) 2) 3)

But what are the values returned by these three function calls? Given that we have the type of add as:

```
val add = fn : (int -> (int -> (int -> int)))
```

The values of the calls to add are as follows:

```
add 1     = fn y => fn z => 1 + y + z
add 1 2   = fn z => 1 + 2 + z
add 1 2 3 = 1 + 2 + 3
```

In summary, the add function shows us an example of what is known as partial application of a curried function, where not all the arguments need to be supplied at once. Functions in ML only take one parameter. While it may be a tuple, such as with the *sum* function, they are still only one argument.

Unification

We want to check if two types are the same
Examples:

- int = int $\implies$ Yes
- int = bool $\implies$ No
- int = α $\implies$ Yes, α can be anything (provided it has not already been constrained).
- int $\to$ bool = int $\to$ bool $\implies$ Yes
- int $\to$ bool = int $\to$ int $\implies$ No
- int $\to$ bool = α $\to$ bool $\implies$ Yes, provided that we can constrain α to an int
- int $\to$ β = α $\to$ bool $\implies$ Yes, provided that α can be constrained to an int and β can be to a bool
- int $\to$ β = α $\to$ α $\implies$ Yes, provided that α and β can both be constrained to ints.

Constraints

If we have

$$\alpha = \text{ int}$$
$$\alpha = \text{ bool}$$

We establish α to be an int by the first rule, then the second term causes an error (int $\neq$ bool). We say that α is constrained to an int.

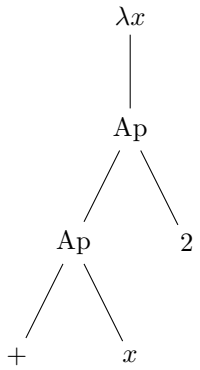
Type inference

We represent our program in terms of constraints. We then pass these constraints to our unification algorithm, which will check if the constraints can be satisfied.

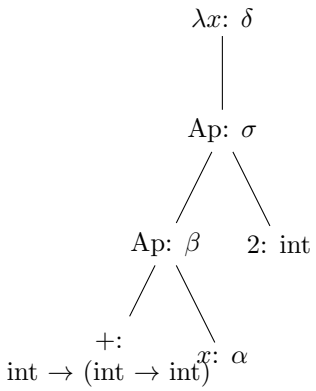
Back to our initial program:

fun f x = x + 2;

The AST for this program will be



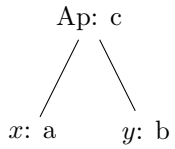
From here, we establish any known types in our tree, and represent the rest of the nodes as unknown variables. Doing this results in the tree:



To go from here, we need to establish some general constraints.

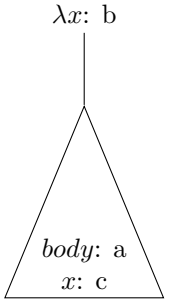
General Constraints

- Given some AST of the form



  We generate the contraint: a = b → c.

- Given some AST of the form



  Given some λ-function λx that contains a body of type a that contains the bound variable x, the resulting type of the functionλx must be a function from x to y. In other words, b = c → a.

Back to the problem

From our tree and the rules that were just defined, we can define the following set of constraints:

$$2 = \text{ int}$$
$$+ = \text{int } \to (\text{int } \to \text{ int})$$
$$x = \alpha$$
$$\text{int } \to (\text{int } \to \text{ int}) = \alpha \to \beta \quad \text{(i)}$$
$$\beta = \text{int } \to \sigma \quad \text{(ii)}$$
$$\delta = \alpha \to \sigma$$

By working our way down, we can make the following observations:

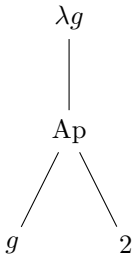
1. From (i), we see that $\alpha = \text{int}$
2. From (i), we see that $\beta = \text{int} \to \text{int}$
3. From (ii) and (2), we see that $\sigma = \text{int}$
4. From (1) and (3), we see that $\delta = \text{int} \to \text{int}$

Therefore, our function has type int → int.

Another example

fun f g = g 2;

This function generates the following tree:



We can get rid of the names (as the textbook does) by establishing a pointer to the unnamed bound variable. We know that 2 has type int, we can further ssume that the lambda-node has type σ, the Ap node has type β, and variable g has type α. We then have he following constraints:

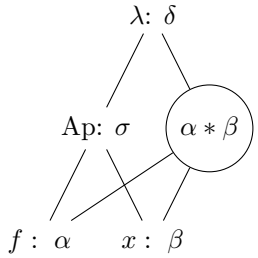
$$2 = \text{ int}$$
$$\alpha = \text{int } \rightarrow \beta$$
$$\sigma = \alpha \rightarrow \beta$$

We can then substitute α into σ to find that our function has type (int $\rightarrow \beta) \rightarrow \beta$.

Third Example

fun apply (f, x) = f x;

From this, we can generate the following AST with types:



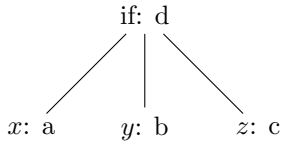
The constraints that can be derived from this are

$$\alpha = \beta \rightarrow \sigma$$
$$\delta = (\alpha * \beta) \rightarrow \sigma$$

Through simplification, we see that the type of this function is $((\beta \rightarrow \sigma) * \beta) \rightarrow \sigma$.

The IF Schema

We can define the behavior of an if-expression through the following schema:



We know that, for an if-expression the first argument must be a bool, and that the second two arguments must be the same. As well, we must return one of the last two values. So, this means that:

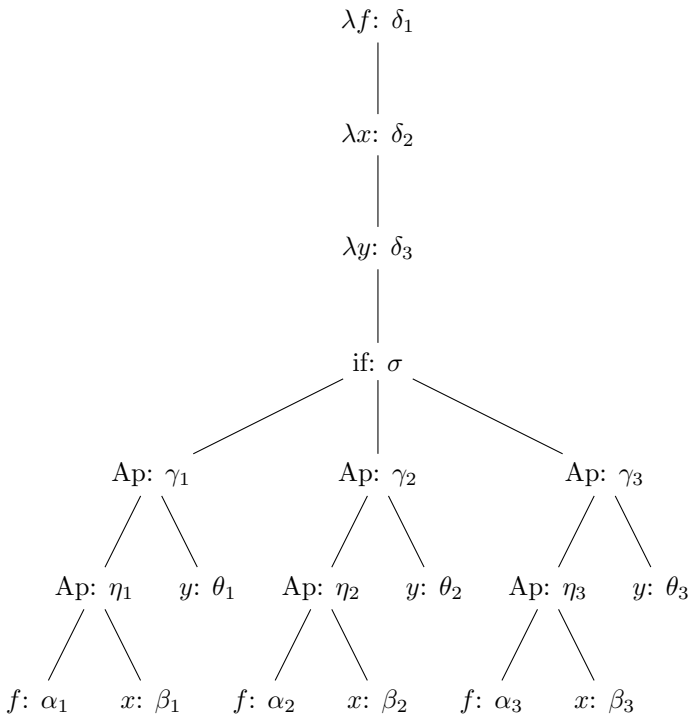
- a = bool
- b = c
- d = b

If-example Example

Given the following code:

```
fun g f x y = if (f x y) then (f 3 y) else (f x "zero");
```

This function generates the following AST with types:



We can establish our defined types:

$$\beta_2 = \text{ int}, \theta_3 = \text{ string}$$

We can see from our application schemas that

$$\alpha_k = \beta_k \to \eta_k \text{ for } k = 1, 2, 3$$
$$\eta_k = \theta_k \to \gamma_k \text{ for } k = 1, 2, 3$$

We also know that $\gamma_1 = \text{bool}$ and $\sigma = \gamma_2 = \gamma_3$, which means that

$$\alpha_1 = \beta_1 \to (\theta_1 \to \text{ bool})$$
$$\alpha_2 = \text{int } \to (\theta_1 \to \gamma_2)$$
$$\alpha_3 = \beta_3 \to (\text{string } \to \gamma_3)$$

We also can establish our λ-constraints:

$$\delta_3 = \theta_k \to \sigma,\ \delta_2 = \beta_k \to \delta_3,\ \delta_1 = \alpha_k \to \delta_2$$

We know, though, since α_k, β_k, θ_k, and γ_k all relate to bound variables, they must all share the same types. So,

$$\beta_k = \text{ int},\ \theta_k = \text{ string},\ \gamma_k = \text{ bool, and } \alpha_k = \text{ int} \to (\text{string} \to \text{bool})$$

We can then chain our λ-constraints to find our result:

$$\begin{gathered} \delta_1 = \alpha_k \to \beta_k \to \theta_k \to \sigma \\ \implies (\text{int } \to (\text{ string } \to \text{ bool})) \to \text{ int } \to \text{ string } \to \text{ bool} \end{gathered}$$