UNIVERSITY OF Southampton
School of Electronics and Computer Science

COMP2212
PROGRAMMING LANGUAGE CONCEPTS

Dr Julian Rathke

TYPE INFERENCE

IMPLICIT TYPE ANNOTATIONS

- You will have noticed that in our Toy language we explicitly declared the type of arguments to functions and local variables
  - λ (x :T) E and let (x :T) = E1 in E2
- This is common practice in many mainstream programming languages - especially statically typed ones (cf. C, C++, Java)
- This is one of the points that advocates of dynamically typed languages often pick on when criticising static typing. It is a burden to the programmer.
- What would be better then is a statically typed language in which the programmer isn't obliged to declare the types of every variable, function, method, etc.
- In this case, the type checker would need to **infer** the types of these entities from their usage in the code.
- For example, `let x = 20 in y + x` , would reasonably allow the type checker to understand that both x and y have type `int` by their usage.
- This is common practice in many functional programming languages. e.g. you have already being doing this in Haskell.

TYPE INFERENCE

- For languages with implicit types, we often refer to the type checking part of compilation as **Type Inference** rather than Type Checking.
- Type Inference is algorithmically more complicated than Type Checking.
- Let's look at the rule for lambda in our Toy languages to see why.

$$\frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \lambda(x : T)E : T \to U}\textsc{TLam}$$

- Suppose instead that we don't know T and U as part of the syntactic definition.
- Then the rule would have to look like this:

$$\frac{\Gamma, x :?? \vdash E : U}{\Gamma \vdash \lambda(x)E :?? \to U}\textsc{TLam}$$

- and algorithmically we would have
- λ `(x) E ->`

```
    check E has some type U in some environment Γ , x : ??
    return type  ?? -> U
  You can see how this complicates things.
```

TYPE VARIABLES AND UNIFICATION

- The approach often taken to solve this problem is to introduce **Type Variables**.
- These are symbolic values that represent an unknown, or unconstrained type.
- When typing a function with unknown types, type variables are used and type checking continues.
- As part of type checking, certain *constraints* on these type variables will arise. e.g. if an argument to a function of unknown type is used as a guard of an IF statement then it must be a boolean.
- So, the type checking algorithm will produce, for each well-typed program, a type that may contain variables, along with a collection of constraints on these type variables.
- To obtain an actual type for the program we need to solve the constraints. That, is we find a substitution of type variables such that all of the constraints hold.
- This latter process is called **unification**.
- This is the basis of type inference in Haskell - there type variables are represented as types written as a , b and c

EXAMPLE OF UNIFICATION

Let's try infer types for this Toy program (with implicit types)

```
let foo = λ(x) if (x < 3) then 0 else (x + 1)
in let cast = λ(y) if (y) then 1 else 0
in  cast (foo (42))
```

Step 1 - unfold the first let.

foo : a , **λ** x : if (x<3) then 0 else (x+1) : a, let cast = ... : b

Step 2 : unfold the **λ**x expression: constraint **a = c -> d** and

if (x<3) then 0 else (x+1) : d assuming x : c .

Step 3 : x < 3 : bool , 0 : int , (x+1) : int this requires x : c to have type int so a constraint **c = Int** is generated.

Also we know **d = Int**

Step 4 : unfold the second let statement :

cast : e , **λ**(y) if (y) then 1 else 0 : e, and cast(foo(42)) : f. This generates the constraint **b = f**

Step 5: unfold the **λ**y expression: constraint **e = g -> h** and if (y) then 1 else 0 : h assuming y : g

Step 6 : unfold the if. y : Bool (so **g = Bool**) and 1 : Int and 0 : Int. This generates the constraint **h = Int**

Step 7: unfold the applications:

cast (foo (42)) : f with the constraint **f = h** and further, by unwinding foo(42) we get **c = Int**.

We can also infer **g=d** because d is the return type of foo.

a = c -> d

c = Int, d = Int,

b = f

e = g -> h

g = Bool h = Int

f = h and g = d

THIS EXAMPLE IS ILL-TYPED