UNIVERSITY OF Southampton
School of Electronics and Computer Science

COMP2212 PROGRAMMING LANGUAGE CONCEPTS

Dr Julian Rathke

TYPE CHECKING

IMPLEMENTING THE TYPE RULES

- In the previous lectures we saw how to specify a type system
  - We gave very precise descriptions of which programs have which types.
  - We used type inference rules to form an inductive relation
- In **this** lecture we will look at how to implement such sets of type rules
  - This is where the pain of using inductive rules pays off for us

- Recall that the typing relation $\Gamma \vdash E : T$ is defined as the smallest relation which satisfies the set of rules.
  - Logically then, if a given program E is in this relation with type T then the only way it got in to the relation was by using one of the rules.
  - But which one ?

SYNTAX DIRECTED RULES

- A set of inference rules S defined over programs used to define an inductive relation R is called **syntax directed** if, whenever a program (think AST) E holds in R then there is a unique rule in S that justifies this. Moreover, this unique rule is determined by the syntactic operator at the root of E.

- For example: in our Toy language this term is well typed and has type Int.

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

- Note that the **last** rule used to derive this fact must have been

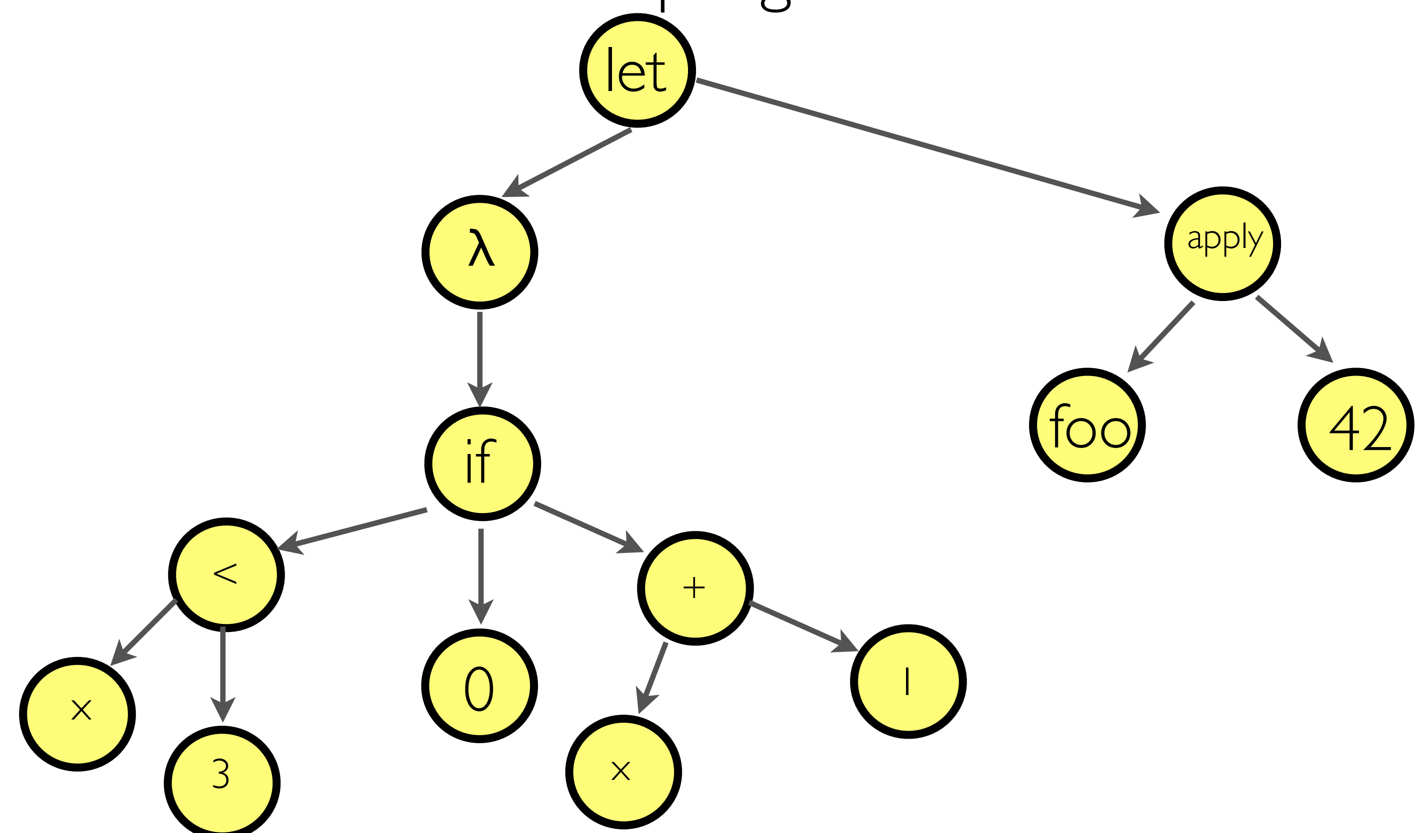
$$\frac{\Gamma \vdash E_1 : T \quad \Gamma, x : T \vdash E_2 : U}{\Gamma \vdash \textsf{let}\ (x : T)\ =\ E_1\ \textsf{in}\ E_2 : U}\textsc{TLet}$$

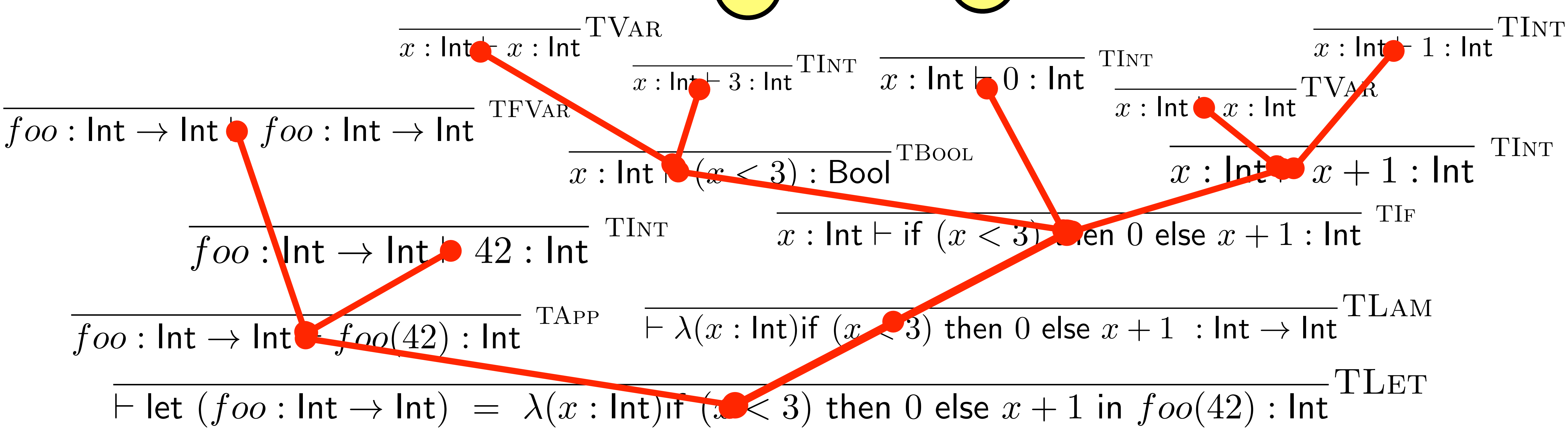
the full derivation would have also required use of TIf ,TLt, TAdd, TLam, TApp, and TInt

STRUCTURE OF TYPE DERIVATION TREES

Interestingly, for a syntax directed set of type rules we see that the structure of type derivation trees matches the structure of the AST of the program that we are deriving a type for.

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





INVERSION LEMMA

- Another important property that we desire of a typing relation is that of **Inversion**.
- This refers to the ability to infer the types of subprograms from the type of the whole program - essentially by reading the type rules from bottom to top.
- Here is the Inversion Lemma for the Toy Language

Lemma (Inversion)

- If $\Gamma \vdash n : T$ then T is Int
- If $\Gamma \vdash b : T$ then T is Bool
- If $\Gamma \vdash x : T$ then $x : T$ is in the mapping Γ
- If $\Gamma \vdash E1 < E2 : T$ then $\Gamma \vdash E1 : Int$ and $\Gamma \vdash E2 : Int$ and T is Bool
- If $\Gamma \vdash E1 + E2 : T$ then $\Gamma \vdash E1 : Int$ and $\Gamma \vdash E2 : Int$ and T is Int
- If $\Gamma \vdash$ if $E1$ then $E2$ else $E3 : T$ then $\Gamma \vdash E1 : Bool$ and $\Gamma \vdash E2 : T$ and $\Gamma \vdash E3 : T$
- If $\Gamma \vdash \lambda (x : T)\ E : U'$ then $\Gamma, x : T \vdash E : U$ and U' is $T \rightarrow U$
- If $\Gamma \vdash$ let $(x : T) = E1$ in $E2 : U$ then $\Gamma, x : T \vdash E2 : U$ and $\Gamma \vdash E1 : T$
- If $\Gamma \vdash E1\ E2 : U$ then $\Gamma \vdash E1 : T \rightarrow U$ and $\Gamma \vdash E2 : T$ for some T

This is easy to prove - but more importantly, yields a direct algorithm for working out types!

A TYPE CHECKING ALGORITHM IN PSEUDOCODE

It's very straightforward to turn this in to Haskell code.

```
Input : term E, environment Γ,
 match E with
  n -> return Int
  b -> return Bool
  x -> look up x in Γ, return its mapped type
  E1 < E2 -> check E1, E2 have type Int, return Bool
  E1 + E2 -> check E1, E2 have type Int, return Int
  if E1 then E2 else E3 -> check E1 has type Bool,
        check E2 and E3 have same type T , return this T
  lambda (x:T)E -> check E has type U in environment Γ, x : T ,
     return type T -> U
  let(x : T) = E1 in E2 ->
    check E1 has type T in environment Γ ,
    check E2 has type U in environment Γ, x : T,  return type U
  E1 E2  -> check E1 has some type T->U,
            check E2 has type T, return type U.
```

NEXT LECTURE: TYPE INFERENCE