UNIVERSITY OF Southampton

School of Electronics and Computer Science

COMP2212
PROGRAMMING LANGUAGE CONCEPTS

Dr Julian Rathke

TYPE RULES FOR TOY

REMINDER : GRAMMAR OF THE TOY LANGUAGE

The grammar for the Toy language we are using is as follows:

```
T , U ::=  Int | Bool
E ::=  n | true | false | E < E | E + E | x |
       | if E then E else E
       | let (x : T) = E in E
```

REMINDER : TYPE DERIVATION RULES AND TREES

The general form of a type derivation rule is

$$\frac{\vdash E_1 : T_1 \qquad \vdash E_2 : T_2 \qquad \ldots \qquad \vdash E_n : T_n}{\vdash E : T}$$

"If the relation holds for the things above the line then the relation holds for things below the line also".

In order to show that $\vdash$ E : T holds, the rules must be formed in to a tree such that the leaf nodes of the tree have no premises. For example

$$\dfrac{\dfrac{\dfrac{}{\vdash E_0 : T_0}}{\vdash E_1 : T_1} \qquad \dfrac{\dfrac{\dfrac{}{\vdash E_4 : T_4} \quad \dfrac{}{\vdash E_5 : T_5}}{\vdash E_3 : T_3}}{\vdash E_2 : T_2}}{\vdash E : T}$$

RULES FOR THE TOY LANGUAGE

Let's write type derivation rules for our toy language, one construct at a time, then.
First, rules for the values:

$$\frac{}{\vdash n : \mathsf{Int}}\ \textsc{TInt} \qquad \frac{}{\vdash b : \mathsf{Bool}}\ \textsc{TBool}$$

It can be useful to give the each type derivation rule a name (cf. TInt and TBool).

Let's look at conditional expressions:

$$\frac{\vdash E_b : \mathsf{Bool} \quad \vdash E_1 : T \quad \vdash E_2 : T}{\vdash \mathsf{if}\ E_b\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 : T}\ \textsc{TIf}$$

These are the key points

TYPING RULES FOR LET EXPRESSIONS

Consider the local variable construct: **`let ( x : T ) = E in E`**

What type does this whole expression have?

As a first guess at a type rule we could write:

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

This is hopelessly wrong!

Environment: map env with type

TYPING EXPRESSIONS WITH FREE VARIABLES

Consider the following example:

```
let ( x : Int ) = 10 in x + 1
```

An instantiation of the broken rule on the previous slide to this example is

$$\frac{\vdash 10 : \mathsf{Int} \qquad \vdash x + 1 : \mathsf{Int}}{\vdash \mathsf{let}\ (x : \mathsf{Int})\ =\ 10\ \mathsf{in}\ x + 1 : U}\ \textsc{TLet}?$$

But how can we show that $\vdash$ x + 1 : Int holds

without knowing anything about x ?

TYPE ENVIRONMENTS

In order to give a type to expression E, we need to have assumptions about the types of the free variables in E. We call these assumptions **Typing Environments** and we use the greek letter Γ to represent them.

Formally, a type environment Γ is a mapping from variable names to Types.

We write entries in this mapping comma-separated

e.g. x : Int , y : Bool , z : Int , ...

A CORRECT RULE FOR LET EXPRESSIONS

Our type relation ⊢ E : T needs to be modified to include the type environment.

We will write $\Gamma \vdash$ E : T to mean, in environment Γ, expression E has type T.

Is this better?

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

Not quite, we need to enlarge the environment in which E_2 is typed

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

This means extend the mapping with a distinct new entry

We will always need to do this for constructs that bind variables.

Think about the scope of x !

Is x in scope in E_2, ? Yes. So, enlarge the type environment with it.

Is x in scope in E_1 ? No. So don't. Unless we want recursive expressions …

NEARLY THERE

We still need a type rule for comparisons:

$$\frac{\Gamma \vdash E_1 : \mathsf{Int} \quad \Gamma \vdash E_2 : \mathsf{Int}}{\Gamma \vdash E_1 < E_2 : \mathsf{Bool}}\ \textsc{TLt}$$

e.g.
a = Int,
where a is x and Int is type T,
Gamma >>
a : Int, tax : float ...
> is a set of variables

so a : T e Gamma
so

In Gamma, a has type T

And a type rule for addition:

$$\frac{\Gamma \vdash E_1 : \mathsf{Int} \quad \Gamma \vdash E_2 : \mathsf{Int}}{\Gamma \vdash E_1 + E_2 : \mathsf{Int}}\ \textsc{TAdd}$$

And type rules for variables:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{TVar}$$

Gamma is the environment (set of variables)

x with type T belongs to the set Gamma

In env Gamma, x is of type T

And that's it. All we need to do now is ask that $\vdash$ is the smallest relation that satisfies all of the type rules for this language. By the induction principle, this defines the relation for every program of the language!