TYPING RULES
HOW DO WE KNOW WHETHER CODE IS WELL-TYPED?

• Most typed languages have some sort of annotation to indicate the type of data. E.g.
  • C asks variables to be declared with a (weak) type so that the compiler can allocate enough memory to store values of that type.
  • Java asks for methods to declare the type of their input parameters as well as their return value
• The way that the compiler makes use of these annotations determines whether the type system is strong/weak.
• We’ll talk about strong (static) type systems for the remainder of this lecture.
• The compiler needs to check that, at any point in a program’s execution, the program doesn’t attempt to consume data of the wrong type.
• Certain operators of each programming language consume data. So the compiler needs to find uses of these operators.
• Clearly then, in order to check well-typedness of programs, this will amount to checks on abstract syntax trees according to the types of the data being used.
A TOY LANGUAGE

Let's play a little game with ASTs for a Toy language:

\[
T, U ::= \text{Int} \mid \text{Bool} \\
E ::= n \mid \text{true} \mid \text{false} \mid E < E \mid E + E \mid x \mid 
\quad \text{if } E \text{ then } E \text{ else } E \\
\quad \text{let } (x : T) = E \text{ in } E
\]

where \( n \) ranges over natural numbers.

Consider the following example and its AST:

\[
\text{let } (x : \text{Int}) = \\
\quad \text{if } (y < 3) \text{ then } 0 \text{ else } (y + 1) \\
\text{in } x + 42
\]
Let's rewrite the previous example with Types in place of the values and value expressions!

```plaintext
let ( Int )
    = if ( Bool ) then Int else Int
in
Int
```

Now, one way of checking types is to traverse this type abstracted AST and see whether the operators that consume data are given data of the correct type.

The operators that consume data are: if-then-else (this reads a Bool), < consumes two Ints and + consumes two Ints.
COMPLICATE THE EXAMPLE

Let’s make it more fun

```plaintext
let (x : Int) = if (if (y < 0) then false else true) then 0 else (y + 1) in x + 42
```

Every subtree in the AST needs to have a type!

Every program fragment needs to have a type!
So is this program well typed? In a dynamically typed language you might say so. What about a static type?

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

So what is the type of this subtree? We can't answer this statically without looking at the rest of the code. We know it is either an Int or Bool - but which one in general depends on the conditional check. This can depend on run time information - e.g. user inputs - even without this, it is undecidable in general.
WHAT HAVE WE LEARNED

• Every program fragment E needs to be given a type T in order to build up a picture of whether the whole AST is well-typed.
  • We need to define the ‘typing relation’ written ⊢ E : T and read ‘E’ has type ‘T’)
  • We need to define this for every possible program E!
• We will want to do local checking on syntax trees
  • The type of a program op ( E1, E2, ..., En ) should depend only on the types of E1, E2, ..., En
  • Only certain operators generate actual checks for correct usage of types
  • We may need to approximate the type where we can’t determine it statically
• To define a relation over the set of all programs is an interesting challenge.
  • But we know something special about the shape of the set of all programs!
  • This is going to help us enormously.
• Let’s try do this for our toy language.
THINGAMYS AND WHATSITS - A BRIEF EXCURSION

• Let suppose we want to build an infinite set of Things: here are some rules for building it
  • You can start with a Doobry or a Whatsit
  • Given anything in the set already, you can put a Thingamy on top of it.
  • Given anything with a Thingamy on top you can add a BlahBlah.
  • The set is the smallest set that you can build using these rules.
• Do we know anything about the shape of this set? Yes, lots.

This is an example of an inductively defined set.
The interesting thing is that, because the resulting set is the smallest such set, and we know all of the constructors for the set, then everything in the set must take one of these shapes.

Let's write a little grammar for the set:

\[ E ::= \text{Doobry} \mid \text{Whatsit} \mid \text{Thingamy } E \mid \text{BlahBlah}(\text{Thingamy } E) \]

So, in order to define a function or relation on this set, it is sufficient to define the function or relation inductively by specifying it on these shapes. e.g.

\[
\begin{align*}
\text{fun BlahCount ( Doobry )} & = 0 \\
\text{fun BlahCount ( Whatsit )} & = 0 \\
\text{fun BlahCount ( Thingamy } E \) & = \text{BlahCount ( } E \) \\
\text{fun BlahCount ( BlahBlah ( Thingamy } E \) ) & = 1 + \text{BlahCount ( } E \)
\end{align*}
\]

and then asking for the smallest function that satisfies these rules.

In order to define a function or relation on the set of all programs of a programming language then, we just need to define it on each program construct of the language.

This is handy because we need to define typing relations \( \vdash E : T \) for all programs in the toy language. We can do this one construct at a time.
The general form of a type derivation rule is

\[
\frac{\vdash E_1 : T_1 \quad \vdash E_2 : T_2 \quad \ldots \quad \vdash E_n : T_n}{\vdash E : T}
\]

This can be read as “If the relation holds for the things above the line then the relation holds for things below the line also”. Sometimes there are no premises above the line.

In general, a programming language will be given a set of such rules. Then, 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

\[
\frac{\vdash E_0 : T_0}{\vdash E_1 : T_1}\quad \frac{\vdash E_3 : T_3}{\vdash E_4 : T_4}\quad \frac{\vdash E_2 : T_2}{\vdash E_5 : T_5}
\]

\[
\frac{\vdash E_1 : T_1}{\vdash E : T}
\]
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:

\[
\begin{align*}
\vdash n : \text{Int} & \quad \text{TInt} \\
\vdash b : \text{Bool} & \quad \text{TBool}
\end{align*}
\]

It can be useful to give the each type derivation rule a name (cf. TInt and TBool). Let's look at conditional expressions:

\[
\begin{align*}
\vdash E_b : \text{Bool} & \quad \vdash E_1 : T & \quad \vdash E_2 : T \\
\vdash \text{if } E_b \text{ then } E_1 \text{ else } E_2 : T & \quad \text{TIf}
\end{align*}
\]

These are the key points
Consider the local variable construct: \( \text{let} \ (x : T) = E \ \text{in} \ E \)

What type does this whole expression have?

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

\[
\frac{\vdash E_1 : T \quad \vdash E_2 : U}{\vdash \text{let} \ (x : T) = E_1 \ \text{in} \ E_2 : U} \quad \text{TLET?}
\]

This is hopelessly wrong!
Consider the following example:

\[
\text{let } (x : \text{Int}) = 10 \text{ in } x + 1
\]

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

\[
\frac{\vdash 10 : \text{Int} \quad \vdash x + 1 : \text{Int}}{\vdash \text{let } (x : \text{Int}) = 10 \text{ in } x + 1 : U} \quad \text{TLET?}
\]

But how can we show that \(\vdash x + 1 : \text{Int}\) holds without knowing anything about \(x\)?
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 $\Gamma$ to represent them.

Formally, a type environment $\Gamma$ is a mapping from variable names to Types. We write entries in this mapping comma-separated

\[ x : \text{Int}, \ y : \text{Bool}, \ z : \text{Int}, \ \ldots \]
A CORRECT RULE FOR LET EXPRESSIONS

Our type relation \( \vdash E : T \) needs to be modified to include the type environment. We will write \( \Gamma \vdash E : T \) to mean, in environment \( \Gamma \), expression \( E \) has type \( T \).

Is this better? \[
\frac{\Gamma \vdash E_1 : T \quad \Gamma \vdash E_2 : U}{\Gamma \vdash \text{let } (x : T) = E_1 \text{ in } E_2 : U} \]

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

\[
\frac{\Gamma \vdash E_1 : T \quad \Gamma, x : T \vdash E_2 : U}{\Gamma \vdash \text{let } (x : T) = E_1 \text{ in } E_2 : U} \]

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:

\[
\frac{\Gamma \vdash E_1 : \text{Int} \quad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 < E_2 : \text{Bool}} \quad \text{TLT}
\]

And a type rule for addition:

\[
\frac{\Gamma \vdash E_1 : \text{Int} \quad \Gamma \vdash E_2 : \text{Int}}{\Gamma \vdash E_1 + E_2 : \text{Int}} \quad \text{TADD}
\]

And type rules for variables:

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{TVAR}
\]

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!
• **Lambda Calculus** can be considered as the simplest “interesting” programming language.
• In its purest form it contains just two primitives: function abstraction and application.
• But, in its untyped form, it is Turing complete - that is, all computation can be expressed in it!
• It is the foundation of most real functional programming languages (OCaml, Haskell, Scheme etc).

The grammar of the language is simply:

\[
E ::= x \ | \ \lambda x. E \ | \ E \ E
\]

\(x\) is drawn from a set of Variables.
SOME EXAMPLE LAMBDA CALCULUS TERMS

The identity function: \( \lambda x . x \) is a function that takes a single argument and simply returns it.

First projection: \( \lambda x . \lambda y . x \) is a function that takes two arguments and returns the first

Second projection: \( \lambda x . \lambda y . y \) is a function that takes two arguments and returns the second

Twice : \( \lambda f . \lambda x . f ( f ( x ) ) \) is a function that takes a single-argument function and an argument and twice applies the supplied function \( f \) to the argument \( x \).

Composition : \( \lambda g . \lambda f . \lambda x . f ( g ( x ) ) \) takes two functions and an argument and returns the composed function \( f ; g \)
Let’s call the lambda expression \( \lambda x . \lambda y. x \) the name \( T \)

Let’s call the lambda expression \( \lambda x . \lambda y. y \) the name \( F \)

We can think of these as “Boolean” values in the lambda calculus!
That is, “Boolean”s are functions that take two arguments (representations of true and false) and return one of them as an answer

What does the following function \( A \) do if you apply it to a “Boolean”? \( \lambda b . \lambda c . b \ c \ b \)

If \( b \) is supplied the \( T \) boolean then it returns \( c \).
So if \( c \) is \( T \) then the whole result is \( T \). If \( c \) is \( F \) then the whole result is \( F \).
If \( b \) is supplied the \( F \) boolean then it returns \( b \) - which is \( F \).

This is Boolean conjunction a.k.a. \( \text{AND} \)

Can we write \( \text{OR} \) ?

Can we write \( \text{NOT} \)?

Easy: \( \lambda b . \lambda c . b \ b \ c \)

Easy: \( \lambda b . \lambda x . \lambda y . b \ y \ x \)
The identity function: \( \lambda x . x \colon T \to T \)

First projection: \( \lambda x . \lambda y . x \colon T \to (U \to T) \)

Second projection: \( \lambda x . \lambda y . y \colon T \to (U \to U) \)

Twice: \( \lambda f . \lambda x . f ( f (x) ) \colon (T \to T) \to T \to T \)

Composition: \( \lambda g . \lambda f . \lambda x . f ( g (x) ) \colon (T \to U) \to (U \to V) \to (T \to V) \)
We can formalise a simply type system for lambda calculus. Without some base types though this is a very uninteresting language so let’s look at it combined with the Toy language. We’ll call the resulting language \( \lambda \text{Toy} \)

\[
T, U ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T
\]

\[
E ::= n \mid \text{true} \mid \text{false} \mid E < E \mid E + E \mid x \mid \text{if } E \text{ then } E \text{ else } E \mid \lambda (x : T) E \mid \text{let } (x : T) = E \text{ in } E \mid E \ E
\]

The type rules for the added constructs are straightforward given what we have learnt already:

\[
\frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \lambda(x : T)E : T \rightarrow U} \quad \text{T\text{LAM}}
\]

\[
\frac{\Gamma \vdash E_1 : T \rightarrow U \quad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \ E_2 : U} \quad \text{T\text{APP}}
\]
THE TYPE SYSTEM IN FULL (FOR REFERENCE)

\[
\begin{align*}
\Gamma \vdash n & : \text{Int} \quad & \text{TInt} \\
\Gamma \vdash b & : \text{Bool} \quad & \text{TBool} \\
\Gamma \vdash x & : T \quad & \text{TVAR} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash E_1 & : \text{Int} \quad \Gamma \vdash E_2 & : \text{Int} \quad & \text{TLT} \\
\Gamma \vdash E_1 < E_2 & : \text{Bool} \\
\Gamma \vdash E_1 & : \text{Int} \quad \Gamma \vdash E_2 & : \text{Int} \quad & \text{TAdd} \\
\Gamma \vdash E_1 + E_2 & : \text{Int} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash E_b & : \text{Bool} \quad \Gamma \vdash E_1 & : T \quad \Gamma \vdash E_2 & : T \quad & \text{TIf} \\
\Gamma \vdash \text{if } E_b \text{ then } E_1 \text{ else } E_2 & : T \\
\Gamma \vdash E_1 & : T \quad \Gamma, x : T \vdash E_2 & : U \quad & \text{TLet} \\
\Gamma \vdash \text{let } (x : T) = E_1 \text{ in } E_2 & : U \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : T \vdash E & : U \quad & \text{TLam} \\
\Gamma \vdash \lambda(x : T)E : T \rightarrow U \\
\Gamma \vdash E_1 : T \rightarrow U \quad \Gamma \vdash E_2 & : T \quad & \text{TApp} \\
\Gamma \vdash E_1 \ E_2 & : U \\
\end{align*}
\]

Notice the $\Gamma$ in the TInt, TBool and TIf rules.

Phew, I’m glad that real programming languages don’t have more constructs than $\lambda$ Toy!
NEXT LECTURE: TYPE CHECKING