TYPE CHECKING AND TYPE INFERENCE
IMPLEMENTING THE TYPE RULES

• In the previous lecture 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.

\[
\text{let foo} = \lambda (x : \text{Int}) . \text{if } (x < 3) \text{ then } 0 \text{ else } (x + 1) \\
\text{in} \\
\text{foo (42)}
\]

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

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

the full derivation would have also required use of \text{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 = \( x : \text{Int} \).
    if \( x < 3 \)
    then 0
    else \( x + 1 \)
in foo (42)
```

```
let
    \( \lambda \)
    if
        <
        x
        3
    
    +
        0
        x
        1
    
in 42
```

```
foo : \text{Int} \rightarrow \text{Int}
foo : \text{Int} \rightarrow \text{Int}
foo : \text{Int} \rightarrow \text{Int}
foo : \text{Int} \rightarrow \text{Int}
foo : \text{Int} \rightarrow \text{Int}
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
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 $\Gamma$
- If $\Gamma \vdash E_1 < E_2 : T$ then $\Gamma \vdash E_1 : \text{Int}$ and $\Gamma \vdash E_2 : \text{Int}$ and $T$ is Bool
- If $\Gamma \vdash E_1 + E_2 : T$ then $\Gamma \vdash E_1 : \text{Int}$ and $\Gamma \vdash E_2 : \text{Int}$ and $T$ is Int
- If $\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : T$ then $\Gamma \vdash E_1 : \text{Bool}$ and $\Gamma \vdash E_2 : T$ and $\Gamma \vdash E_3 : T$
- If $\Gamma \vdash \lambda (x : T) E : U'$ then $\Gamma, x : T \vdash E : U$ and $U'$ is $T \to U$
- If $\Gamma \vdash \text{let } (x : T) = E_1 \text{ in } E_2 : U$ then $\Gamma, x : T \vdash E_2 : U$ and $\Gamma \vdash E_1 : T$
- If $\Gamma \vdash E_1 ( E_2 ) : U$ then $\Gamma \vdash E_1 : T \to U$ and $\Gamma \vdash E_2 : T$ 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

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.
You will have noticed that in our Toy language we explicitly declared the type of
arguments to functions and local variables

\( \lambda (x : T) \ E \) and let \((x : T) = E_1\) in \(E_2\)

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 OCaml.
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.

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

```
\Gamma, x : T \vdash E : U
\Gamma \vdash \lambda(x : T)E : T \rightarrow U
```

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:

\[
\frac{\Gamma, x : ?? \vdash E : U}{\Gamma \vdash \lambda(x)E : ?? \rightarrow U}^{\text{TLAM}}
\]

```
\Gamma, x : ?? \vdash E : U
\Gamma \vdash \lambda(x)E : ?? \rightarrow U
```

and algorithmically we would have

\[
\lambda(x) E \rightarrow
\]

```
check E has some type U in some environment \( \Gamma, x : ?? \)
return type ?? \rightarrow U
```

You can see how this complicates things.
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)

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

Step 1- unfold the first let.

\(\text{foo} : a, \ \lambda x : \text{if}(x<3)\text{then}0\text{else}(x+1) : a, \ \text{let \text{cast} = ... : b}\)

Step 2 : unfold the \(\lambda x\) expression: constraint \(a = c -> d\) and
  \(\text{if}(x<3)\text{then}0\text{else}(x+1) : d\) assuming \(x : c\).

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

Also we know \(d = \text{Int}\)

Step 4 : unfold the second let statement :

\(\text{cast} : e, \ \lambda y : \text{if}(y)\text{then}1\text{else}0 : e, \ \text{and \text{cast}(foo(42))} : f. \ \text{This generates the constraint} \ b = f\)

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

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

Step 7: unfold the applications:

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

We can also infer \(g=d\) because \(d\) is the return type of \text{foo}.  

\(a = c -> d\)

\(c = \text{Int, d} = \text{Int}, \ b = f\)

\(e = g -> h\)

\(g = \text{Bool}, h = \text{Int}\)

\(f = \text{‘h and g = d}\)

Oh oh.

THIS EXAMPLE IS ILL-TYPED
NEXT LECTURE: INTRODUCTION TO SEMANTICS