Type Inference

Type inference

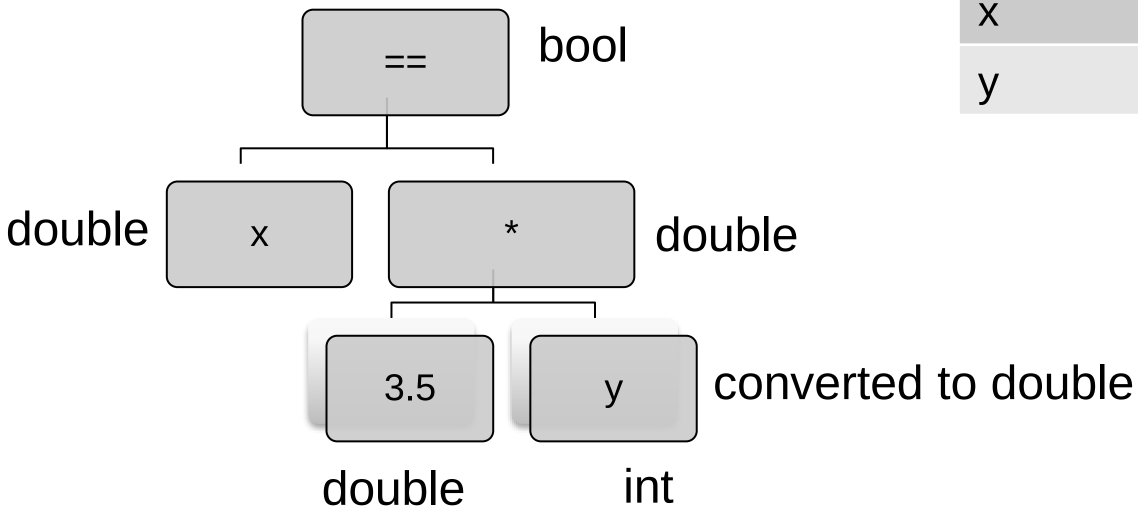
Type checking an expression in the AST usually consists of two steps

1. *Infer the type of the expression*
2. *Check whether the type matches what we expected*

- Now, let's look at step 1
- We can do that starting from the leaves of the AST
- Example: a Boolean expression

x == 3.5 * y

identifier	type
x	double
y	int



Formal Notation

- Type inference can become complex because of type conversions, subtypes, ...
- If you design a new language, it can help that you write down in a formal way what the type rules of the language are
- In the following, we will use this notation:

$$\frac{preconditions}{postconditions}$$

and we write

$$\vdash e : T$$

to say that we are able infer that expression e has type T

Our starting point for the type inference

- What we know (without any preconditions) for example in Java:

$$\frac{}{\vdash \mathit{any\ integer\ constant} : \mathit{int}}$$

$$\frac{}{\vdash \mathit{true} : \mathit{bool}}$$

$$\frac{}{\vdash \mathit{false} : \mathit{bool}}$$

- What if the expression is an identifier?

$$\frac{\begin{array}{c} x \text{ } \mathit{is\ an\ identifier} \\ \mathit{the\ symbol\ table\ says\ that\ the\ type\ of\ } x \mathit{\ is\ } T \end{array}}{\vdash \; x : T}$$

More complex inference rules

- We can now build more complex inference rules:

$$\frac{\vdash e_1 : int, \vdash e_2 : int}{\vdash \; e_1 + e_2 \; : int}$$

- And we can even formulate more general rules:

$$\frac{\vdash e_1 : T, \vdash e_2 : T}{\vdash \; e_1 == e_2 \; : bool}$$

- For function calls:

$$\frac{\begin{array}{c} f \; is \; an \; identifier \\ f \; has \; type \; (T_1, \dots, T_n) \rightarrow T \; in \; the \; symbol \; table \\ \vdash e_1 : T_1, \dots, \vdash e_n : T_n \end{array}}{\vdash \; f(e_1, \dots, e_n) \; : T}$$

Rules for type compatibility

- We can also describe how type compatibility works in a programming language
- We write $A \leq B$ if A is a subtype of B
- For example, in Java, $A \leq B$ if A is a sub-class of B.

For a function call, we can then write this rule:

$$\frac{\begin{array}{c} f\ is\ an\ identifier \\ f\ has\ type\ (T_1, \ldots, T_n) \rightarrow T\ in\ the\ symbol\ table \\ \vdash e_1 : T_1' \leq T_1, \ldots, \vdash e_n : T_n' \leq T_n \end{array}}{\vdash\ \ f(e_1, \ldots, e_n) : T}$$

A better way to think about type errors

- With these rules we can now define more precisely what a type error in the first step (type inference) of type checking is:

type error during inference = none of the inference rules works for the expression, i.e, we cannot infer its type

Function Overloading

- Some languages allow function overloading
- Example: Two functions

```
void f(int x) { ... }
void f(double x) { ... }
```

- If you have a function call like `f(3)` which of the two functions is called, knowing that `int` can be converted to `double`?
- Different languages have different rules for overloaded functions, for example “choose the function with the nearest type”
- Can become difficult for functions with more than one parameter
  - Which function is called in `f(3,3)` ?

```
void f(int x, double y) { ... }
void f(double x, int y) { ... }
```

- Even more complex for OO languages with inheritance etc.

Conclusions

- Designing a type system is hard. It's easy to forget a special case or to design a system that contains inconsistencies
- Don't believe me? Think about all the special cases in programming languages that we have not discussed here:
  - The special role of `null` in Java
  - Generics in Java
  - Multiple inheritance in C++
- You have a free afternoon? Check Java's rules for type inference: https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.7
- Writing inference rules of the form $\frac{preconditions}{postconditions}$ can help you to find problems before you start writing code for the compiler
  - And it allows to document the rules of a language in a formal way (instead of saying "look at the code of the compiler")

Bonus slide: Just for your information...

- In some languages, you don’t have to declare the types of variables and functions. The compiler can infer types automatically
- Example Java (only works for local variables):

```
var i = 10;
```

  → Java compiler infers that `i` has type `int`
- Example Haskell:

```
x = map length [ “abc”, “bde”, “cfg”]
```

  → Haskell compiler infers that `x` has the type `[int]` (list of int) because
    - `[“abc”, “bde”, “cfg”]` is a list of strings
    - `length` is a function $string \rightarrow int$
    - `map` takes a function and applies it to the elements of a list and returns a new list with the results
- As you can see at the example of Haskell, type inference can go very far