

## Topic 6: Types

COS 320

### Compiling Techniques

Princeton University  
Spring 2017

Prof. David August

1

## Defining a Type System

- RE → Lexing
- CFG → Parsers
- Inductive Definitions → Type Systems

An inductive definition really has two parts:

1. Specification of the form of *judgments* – A judgment is an assertion/claim, may or may not be true. A *valid judgment* is a true/provable judgment.
2. A collection of *inference rules* – what allow you to conclude whether a judgment is true or false

3

## Types

What is a type?

Type Checking:

- Helps find language-level errors:
  - Memory Safety – can't dereference something not a pointer
  - Control-Flow Safety – can't jump to something not code
  - Type Safety – redundant specification checked
- Helps find application-level errors:
  - Ensures isolation properties
- Helps generate code:
  - Is that "+" a floating point add or an integer add?

2

## Inference Rules

An implementation-language-independent way: type system with **inference rules**.

$$\frac{a : \text{bool} \quad b : \text{bool}}{a \ \&\& \ b : \text{bool}}$$

Read: if  $a$  has type  $\text{bool}$  and  $b$  has type  $\text{bool}$ , then  $a \ \&\& \ b$  has type  $\text{bool}$ .

4

## Inference Rules

An inference rule has a set of **premises**  $J_1, \dots, J_n$  and one **conclusion**  $J$ , separated by a horizontal line:

$$\frac{J_1 \dots J_n}{J}$$

Read:

- If I can establish the truth of the premises  $J_1, \dots, J_n$ , I can conclude:  $J$  is true.
- To check  $J$ , check  $J_1, \dots, J_n$ .

An inference rule with no premises is called an **Axiom** –  $J$  always true

5

## Axioms and Rules

Examples: BT, BF, B&&, B||

7

## Judgments

The premises and conclusions are called **judgments**.

The most common judgments in type systems have the form:

$$e : T$$

Read: expression  $e$  has type  $T$ .

Means: Based on no outside evidence,  $e$  is an expression with type  $T$

6

## Type Checking and Type Inference

Two activities:

- Type checking: Given an expression  $e$  and a type  $T$ , decide if  $e : T$
- Type inference: Given an expression  $e$ , find a type  $T$  such that  $e : T$

Both activities necessary. Both originate from typing rules.

8

## Type Checking Implementation

Example: type checking for `&&`:

```
check(a && b, bool):  
    check(a, bool)  
    check(b, bool)
```

No patterns matching types other than *bool*.

9

## Recall Symbol Table, Scope Topic

- Generally, a variable can be any type available in the language.
- In C and Java, type determined by the declaration of the variable.
- In inference rules, variables are collected to a context.
- Context is a symbol table of (variable, type) pairs.
- In inference rules, the context is denoted by the Greek letter  $\Gamma$ , Gamma.
- The judgment form for typing is generalized to:

$$\Gamma \vdash e : T$$

- Read: expression *e* has type *T* in context  $\Gamma$

11

## Type Inference Implementation

Example: type checking for `&&`:

```
infer(a && b):  
    check(a, bool)  
    check(b, bool)  
    return bool
```

Inference involves checking.

10

## Context

Consider:

$$x : \text{int}, y : \text{int} \vdash x + y > y : \text{bool}$$

This means:

*x + y > y* is bool in context where *x* and *y* are ints

Context notation:

$$x_1 : T_1, \dots, x_n : T_n$$

Adding variable to existing context:

$$\Gamma, x : T$$

12

## Context

Most judgments share the same  $\Gamma$ , because the context doesn't change.

$$\frac{\Gamma \vdash a : \text{bool} \quad \Gamma \vdash b : \text{bool}}{\Gamma \vdash a \&\& b : \text{bool}}$$

For declarations:

$$\frac{}{\Gamma \vdash x : T} \text{ if } x : T \text{ in } \Gamma$$

The condition “if  $x : T$  in  $\Gamma$ ” is not a judgment – but a sentence in the metalanguage (English). (Condition is a symbol table lookup of  $x$  in  $\Gamma$ .)

13

## Proofs

Proof Tree: a trace of the steps that the type checker performs, built up rule by rule.

$$\frac{\frac{x : \text{int}, y : \text{int} \vdash x : \text{int} \quad x : \text{int}, y : \text{int} \vdash y : \text{int}}{x : \text{int}, y : \text{int} \vdash x+y : \text{int}} + \quad \frac{x : \text{int}, y : \text{int} \vdash y : \text{int}}{x : \text{int}, y : \text{int} \vdash x+y > y : \text{bool}} >}{x : \text{int}, y : \text{int} \vdash x+y > y : \text{bool}}$$

Each judgment is a conclusion from the ones above with some of the rules, indicated beside the line. This tree uses the variable rule and the rules for  $+$  and  $>$ :

$$\frac{}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash a : \text{int} \quad \Gamma \vdash b : \text{int}}{\Gamma \vdash a+b : \text{int}} + \quad \frac{\Gamma \vdash a : \text{int} \quad \Gamma \vdash b : \text{int}}{\Gamma \vdash a > b : \text{bool}} >$$

15

## Functions

Function Application:

$$\frac{\Gamma \vdash a_1 : T_1 \quad \dots \quad \Gamma \vdash a_n : T_n}{\Gamma \vdash f(a_1, \dots, a_n) : T} \text{ if } f : (T_1, \dots, T_n) \rightarrow T \text{ in } \Gamma$$

Notation:

$$(T_1, \dots, T_n) \rightarrow T$$

14

## Overloading

The binary arithmetic operations ( $+$   $-$   $*$   $/$ ) and comparisons ( $==$   $!=$   $<$   $>$   $<=$   $>=$ ) are overloaded in many languages.

If the possible types are `int`, `double`, and `string`, the typing rules become:

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : t}{\Gamma \vdash a + b : t} \text{ if } t \text{ is int or double or string}$$

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : t}{\Gamma \vdash a == b : \text{bool}} \text{ if } t \text{ is int or double or string}$$

16

## Overloading Implementation

First infer the type of the first operand, then check the second operand with respect to this type:

```
infer (a + b) :  
  t := infer(a)  
  // check that t ∈ {int, double, string}  
  check (b, t)  
  return t
```

17

## Type Conversion

$$\frac{\Gamma \vdash a : t \quad \Gamma \vdash b : u}{\Gamma \vdash a + b : \max(t, u)} \text{ if } t, u \in \{\text{int}, \text{double}, \text{string}\}$$

`int < double < string`  
`max(int, string) = string`

2 + "hello" produces "2hello"

Evaluate: 1 + 2 + "hello" + 1 + 2

19

## Type Conversion

Example: an integer can be converted into a double

Generally, integers and doubles have different binary representations operated upon by different instructions.

Compiler generates a conversion instruction (or instructions) for type conversions.

18

## Statement Validity

When type-checking a statement, simply check whether the statement is **valid**.

A new judgment form:

$$\Gamma \vdash s \text{ valid}$$

Read: Statement *s* is valid in environment  $\Gamma$ .

Example: **while**

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s \text{ valid}}{\Gamma \vdash \text{while } (e) \text{ } s \text{ valid}}$$

20

## Expression Statements

Some expressions simply need a type inference.  
For example: assignments and function calls.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e; \text{valid}}$$
$$\frac{x_1 : T_1, \dots, x_m : T_m \vdash s_1 \dots s_n \text{ valid}}{T f(T_1 x_1, \dots, T_m x_m) \{s_1 \dots, s_n\} \text{ valid}}$$

Parameters of the function define the context.  
The body statements  $s_1 \dots s_n$  are checked in this context.  
Context may change within the body from declarations.  
Check all variables in the parameter list are distinct.

21

## Declarations and Block Structure

Each declaration has a scope, in a certain block.

Blocks in C and Java correspond (roughly) to parts of code between curly brackets: `{ }`

Two principles regulate the use of variables:

1. A variable declared in a block has its scope till the end of that block.
2. A variable can be declared again in an inner block, but not otherwise.

23

## Return Statements

Return statement should be of expected type.

Control flow makes this interesting:

```
if (fail()) return 1 ; else return 0 ;
```

22

## Declarations and Block Structure

```
{
  int x ;
  {
    x = 3 ;      // x : int
    double x ;  // x : double
    x = 3.14 ;
    int z ;
  }
  x = x + 1 ;    // x : int, receives the value 3 + 1
  z = 8 ;        // ILLEGAL! z is no more in scope
  double x ;     // ILLEGAL! x may not be declared again
  int z ;        // legal, since z is no more in scope
}
```

24

## Stack of Contexts

---

Context to deal with blocks: Instead of a simple lookup table,  $\Gamma$  must be a stack of lookup tables.

Notation:

$$\Gamma_1.\Gamma_2$$

where  $\Gamma_1$  is an old (i.e. outer) context and  $\Gamma_2$  an inner context.

The innermost context is the top of the stack.

Recall Symbol Table Discussion...

25

## Example: If Statement Derivation/Proof

---

27

## Declarations

---

A declaration introduces a new variable in the current scope, checked to be fresh with respect to the context.

Rules for sequences of statements, not just individual statements:

$$\Gamma \vdash s_1 \dots s_n \text{ valid}$$

A declaration extends the context used for checking the statements that follow:

$$\frac{\Gamma, x : T \vdash s_2 \dots s_n \text{ valid}}{\Gamma \vdash T x; s_2 \dots s_n \text{ valid}} \quad x \text{ not in the top-most context in } \Gamma$$

26

## Example: sizeof

---

28

## Example: Function

---