

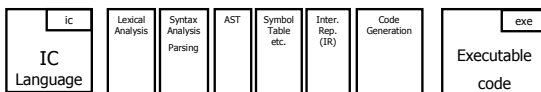
# Winter 2007-2008 Compiler Construction T7 – semantic analysis part II type-checking

Mooly Sagiv and Roman Manevich  
School of Computer Science  
Tel-Aviv University

## Announcements

2

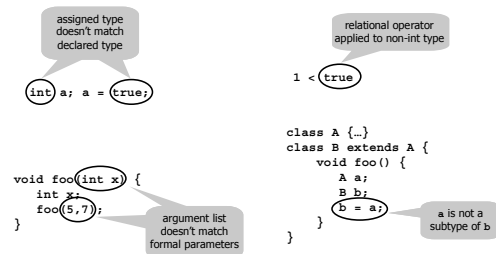
## Today



- Today:
  - Static semantics
    - Creating types
    - Type-checking
  - Overall flow of semantic analysis
- Next time:
  - PA3 – semantic analysis (T6 + T7)

3

## Examples of type errors



4

## Types and type safety

- A type represents a set of values computed during the execution of a program
  - `boolean = {true, false}`
  - `int = {-232, 232}`
  - `void = {}`
- Type safety means that types are used "consistently"
  - Consistency formally defined by typing rules
- Checking type-safety: bind types, check
- Type bindings define type of constructs in a program
  - Explicit bindings: `int x;`
  - Implicit bindings: `x = 1;`
- Check: determine correct usage of type bindings according to typing rules

5

## Class hierarchy for types

```

abstract class Type {...}

class IntType extends Type {...}

class BoolType extends Type {...}

class ArrayType extends Type {
    Type elemType;
}

class MethodType extends Type {
    Type[] paramTypes;
    Type returnType;
    ...
}

class ClassType extends Type {
    ICClass classAST;
    ...
}
    
```

6

## Type comparison

- Option 1: use a unique object for each distinct type
  - Resolve each type expression to same object
  - Use reference equality for comparison (==)
- Option 2: implement a method `t1.equals(t2)`
  - Perform deep (structural) test
- For object-oriented languages also need sub-typing: `t1.subtypeof(t2)`

7

## Creating type objects

- Can build types while parsing
 

```
non terminal Type type;
type ::= BOOLEAN
      | ARRAY LB type:t RB
      | { : RESULT = TypeTable.boolType; : }
      | { : RESULT = TypeTable.arrayType(t); : }
```
- Type objects = AST nodes for type expressions
- Store types in global type table
- When a class is defined
  - Add entry to symbol table
  - Add entry to global type table

8

## Type table implementation

```
class TypeTable {
    // Maps element types to array types
    private Map<Type,ArrayType> uniqueArrayTypes;
    private Map<String,ClassType> uniqueClassTypes;

    public static Type boolType = new BoolType();
    public static Type intType = new IntType();
    ...

    // Returns unique array type object
    public static ArrayType arrayType(Type elemType) {
        if (uniqueArrayTypes.containsKey(elemType)) {
            // array type object already created - return it
            return uniqueArrayTypes.get(elemType);
        }
        else {
            // object doesn't exist - create and return it
            ArrayType arrt = new ArrayType(elemType);
            uniqueArrayTypes.put(elemType,ArrayType);
            return arrt;
        }
    }
    ...
}
```

9

## Type judgments

- Static semantics = formal notation which describes type judgments
 
$$E : T$$
 means "E is a well-typed expression of type T"
- Examples:
  - 2 : int
  - 2 \* (3 + 4) : int
  - true : bool
  - "Hello" : string

10

## Type judgments

- Need to account for symbols
- More general notation:
 
$$A \vdash E : T$$
 means "In the context A the expression E is a well-typed expression with type T"
- Type context = set of type bindings `id : T` (symbol table)
- Examples:
  - `b:bool, x:int`  $\vdash$  `b:bool`
  - `x:int`  $\vdash$  `1 + x < 4:bool`
  - `foo:int->string, x:int`  $\vdash$  `foo(x) : string`

11

## Typing rules for expressions

- Rules notation
  - "If E1 has type int and E2 has type int, then E1 + E2 has type int"

$$\frac{A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{int}} \text{[+]}$$

optional rule name

- No premises = axiom (AST leaves)

$$\frac{}{A \vdash \text{true} : \text{bool}} \quad \frac{}{A \vdash \text{false} : \text{bool}}$$

$$\frac{}{A \vdash \text{int-literal} : \text{int}} \quad \frac{}{A \vdash \text{string-literal} : \text{string}}$$

12

## Some IC expression rules 1

$$\frac{}{A \vdash \text{true} : \text{bool}} \quad \frac{}{A \vdash \text{false} : \text{bool}}$$

$$\frac{}{A \vdash \text{int-literal} : \text{int}} \quad \frac{}{A \vdash \text{string-literal} : \text{string}}$$

$$\frac{A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 \text{ op } E_2 : \text{int}} \quad \text{op} \in \{ +, -, /, *, \% \}$$

$$\frac{A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 \text{ rop } E_2 : \text{bool}} \quad \text{rop} \in \{ <=, <, >, >= \}$$

$$\frac{A \vdash E_1 : T \quad A \vdash E_2 : T}{A \vdash E_1 \text{ rop } E_2 : \text{bool}} \quad \text{rop} \in \{ ==, != \}$$

13

## Some IC expression rules 2

$$\frac{A \vdash E_1 : \text{bool} \quad A \vdash E_2 : \text{bool}}{A \vdash E_1 \text{ lop } E_2 : \text{bool}} \quad \text{lop} \in \{ \&\&, || \}$$

$$\frac{A \vdash E_1 : \text{int}}{A \vdash - E_1 : \text{int}} \quad \frac{A \vdash E_1 : \text{bool}}{A \vdash ! E_1 : \text{bool}}$$

$$\frac{A \vdash E_1 : T[]}{A \vdash E_1.length : \text{int}} \quad \frac{A \vdash E_1 : T[] \quad A \vdash E_2 : \text{int}}{A \vdash E_1[E_2] : T} \quad \frac{A \vdash E_1 : \text{int}}{A \vdash \text{new } T[E_1] : T[]}$$

$$\frac{A \vdash T \in C}{A \vdash \text{new } T() : T} \quad \frac{\text{id} : T \in A}{A \vdash \text{id} : T}$$

14

## Meaning of rules

- Inference rule says: given that antecedent judgments are true then consequent judgment is true

$$\frac{A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{int}} \quad [+]$$

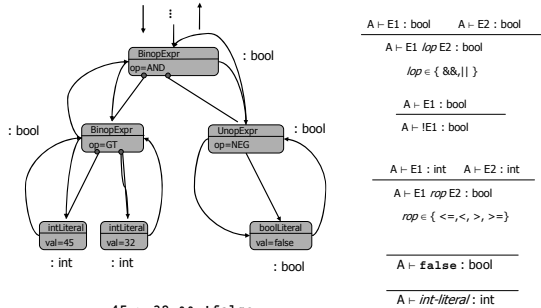
15

## Type-checking algorithm

- Construct types
  - Add basic types to type table
  - Traverse AST looking for user-defined types (classes, methods, arrays) and store in table
  - Bind all symbols to types
- Traverse AST bottom-up (using visitor)
  - For each AST node find corresponding rule (there is only one for each kind of node)
  - Check if rule holds
    - Yes:** assign type to node according to consequent
    - No:** report error

16

## Algorithm example



17

## Type-checking visitor

```
class TypeChecker implements PropagatingVisitor<Type, SymbolTable> {
    public Type visit(ArithBinopExp e, SymbolTable symtab) throws Exception {
        Type lType = e.left.accept(this, symtab);
        Type rType = e.right.accept(this, symtab);
        if (lType != TypeTable.intType())
            throw new TypeError("Expecting int type, found " +
                lType.toString(), e.getLine());
        if (rType != TypeTable.intType())
            throw new TypeError("Expecting int type, found " +
                rType.toString(), e.getLine());
        // we only get here if no exceptions were thrown
        e.type = TypeTable.intType();
    }
    ...
}
```

18

## Statement rules

- Statements have type `void`
- Judgments of the form  $A \vdash S$ 
  - In environment  $A$ ,  $S$  is well-typed
- Rule for `while` statement:

$$\frac{E \vdash e:\text{bool} \quad E \vdash S}{E \vdash \text{while } (e) S}$$

19

## Checking return statements

- Use special entry  $\{\text{ret}:T_r\}$  to represent method return value
  - Add entry to symbol table when entering method
  - Lookup entry when we hit return statement

$$\frac{E \vdash e:T \quad \text{ret}:T' \in E \quad \text{T subtype of } T'}{E \vdash \text{return } e;}$$

$$\frac{\text{ret}:\text{void} \in E}{E \vdash \text{return};}$$

20

## Subtyping

- Inheritance induces subtyping relation
  - Type hierarchy is a tree (anti-symmetric relation)
  - Subtyping rules:

$$\frac{A \text{ extends } B \{...\}}{A \leq B} \quad \frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C} \quad \frac{}{\text{null} \leq A}$$

- Subtyping does not extend to array types
  - A subtype of  $B$  then  $A[]$  is not a subtype of  $B[]$

21

## Type checking with subtyping

- "A value of type  $S$  may be used wherever a value of type  $T$  is expected"
  - $S \leq T \Rightarrow \text{values}(S) \subseteq \text{values}(T)$
- Subsumption rule connects subtyping relation and typing judgments

$$\frac{A \vdash E : S}{S \leq T} \quad \frac{}{A \vdash E : T}$$

- "If expression  $E$  has type  $S$ , it also has type  $T$  for every  $T$  such that  $S \leq T$ "

22

## IC rules with subtyping

$$\frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1 \quad \text{op} \{=, \neq\}}{E \vdash e_1 \text{ op } e_2 : \text{bool}}$$

Method invocation rules:

$$\frac{E \vdash e_0 : T_1 \times \dots \times T_n \rightarrow T_r \quad E \vdash e_i : T_i' \quad T_i' \leq T_i \text{ for all } i = 1..n}{E \vdash e_0(e_1, \dots, e_n) : T_r}$$

$$\frac{(m : \text{static } T_1 \times \dots \times T_n \rightarrow T_r) \in C \quad E \vdash e_i : T_i' \quad T_i' \leq T_i \text{ for all } i = 1..n}{E \vdash C.m(e_1, \dots, e_n) : T_r}$$

23

## More rules

$$\frac{A \vdash S_1 \quad A \vdash S_2; \dots; S_n}{A \vdash S_1; \dots; S_n} \text{ [sequence]}$$

$$\frac{E, x_1:t_1, \dots, E, x_n:t_n, \text{ret}:t_r \vdash S_{\text{body}}}{E \vdash t_r \text{ m}(t_1 \ x_1, \dots, t_n \ x_n) \{S_{\text{body}}\}} \text{ [method]}$$

$$\frac{\text{methods}(C) = \{m_1, \dots, m_k\} \quad \text{Env}(C, m_i) \vdash m_i \text{ for all } i = 1..k}{\vdash C} \text{ [class]}$$

$$\frac{\text{classes}(P) = \{C_1, \dots, C_n\} \quad \vdash C_i \text{ for all } i = 1..n}{\vdash P} \text{ [program]}$$

24

## Semantic analysis flow

- Parsing and AST construction
  - Combine library AST with IC program AST
- Construct and initialize global type table
- Phase 1: Symbol table construction
  - Construct class hierarchy and check that hierarchy is a tree
  - Construct remaining symbol table hierarchy
  - Assign enclosing-scope for each AST node
- Phase 2: Scope checking
  - Resolve names
  - Check scope rules using symbol table
- Phase 3: Type checking
  - Assign type for each AST node
- Phase 4: Remaining semantic checks
- Disclaimer: This is only a suggestion

25

## PA3

- After semantic analysis phase should handle only legal programs (modulo 2 unchecked conditions)
- Note 1: some semantic conditions not specified in exercise (e.g., shadowing fields) – see IC specification for all conditions
- Note 2:

```
class A { A foo() {...} }  
class B extends A { B foo() {...} }
```

Illegal in IC (no support for downcasting)

26

See you next week

27