CS 412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 10: Static Semantics

Administration

- Programming Assignment 2 due in 1 week

Static Semantics

- Can describe the types used in a program. How to describe type checking?
- Formal description: *static semantics* for the programming language
- Is to type-checking as grammar is to parsing
- Static semantics defines types for all legal language ASTs
- We will write ordinary language syntax to mean the corresponding AST

Type Judgements

- Static semantics defines how to derive *type judgments*

E : T means "E is a well-typed expression of type T"

2 : int

2 * (3 + 4) : int

true : bool

"Hello" : string

if (b) 2 else 3 : int

Deriving a judgment

if (b) 2 else 3 : int

- What do we need to decide that this is a well-typed expression of type **int**?

- b must be an bool (b: bool)
- 2 must be an int (2: int)
- 3 must be an int (3: int)

Type Judgments

- Type judgment: $A \vdash E : T$
  - means "In the environment A (symbol table), the expression E is a well-typed expression with the type T"
- Environment is set of *id* : *T*

{b: bool, x: int} ⊢ b : bool

{b: bool, x: int} ⊢ if (b) 2 else x : int

{} ⊢ 2 + 2 : int

Deriving a judgement

- To show

  {b: bool, x: int} ⊢ if (b) 2 else x : int

- Need to show:

  {b: bool, x: int}⊢b: bool

  {b: bool, x: int}⊢2: int

  {b: bool, x: int}⊢x: int

General Rule

- For *any* environment A, expression E, statements S_1 and S_2, the judgment

  $A \vdash \text{if (E) } S_1 \text{ else } S_2 : T$

  is true if

  $A \vdash E : \text{bool}$

  $A \vdash S_1 : T$

  $A \vdash S_2 : T$

As an Inference Rule

Premises

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S_1 : T \qquad A \vdash S_2 : T}{A \vdash \text{if (E) } S_1 \text{ else } S_2 : T} \ (\text{name})$$

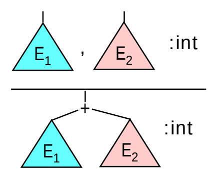
Conclusion

- Holds for any choice of the syntactic variables E, S_1, S_2, T

Meaning of Inference Rule

- Inference rule says: given that antecedent judgments are true
  - with some substitution for meta-variables A, E_1, E_2
- Then, consequent judgment is true
  - with a consistent substitution

$$\frac{A \vdash E_1 : \textbf{int} \qquad A \vdash E_2 : \textbf{int}}{A \vdash E_1 + E_2 : \textbf{int}} \ (+)$$



Implementing a rule

- Work backward from goal:

$T = E\text{.typeCheck}(A) \Leftrightarrow A \vdash E : T$

```
class Add extends Expr {
   Expr e1, e2;
   Type typeCheck(SymTab A) {
        Type   t1 = e1.typeCheck(A),
                  t2 = e2.typeCheck(A);
        if (t1 == Int && t2 == Int) return Int;
        else throw new TypeCheckError("+");
   }
```

$$\frac{A \vdash E_1 : \textbf{int} \qquad A \vdash E_2 : \textbf{int}}{A \vdash E_1 + E_2 : \textbf{int}} \ (+)$$

Why inference rules?

- Inference rules: compact, precise language for specifying static semantics (can specify Java in ~10 pages vs. 100's of pages of Java Language Specification)
- Inference rules correspond directly to recursive AST traversal that implements them
- Type checking is attempt to prove type judgments $A \vdash E : T$ true by walking backward through rules

Proof = Call Graph

let A1 = {b: bool, x: int}

$$\dfrac{\dfrac{A1 \vdash b : bool}{A1 \vdash !b : bool} \quad \dfrac{A1 \vdash 2 : int \quad A1 \vdash 3 : int}{A1 \vdash 2+3 : int} \quad A1 \vdash x : int}{\{ b: bool, x: int\} \vdash \text{if (!b) 2+3 else x} : int}$$

More about Inference Rules

- Rules are implicitly universally quantified over free variables
- No premises: *axiom* $\dfrac{}{A \vdash \text{true} : \textbf{bool}}$
- Same goal judgment may be provable in more than one way:

$$\dfrac{A \vdash E_1 : \textbf{float} \quad A \vdash E_2 : \textbf{float}}{A \vdash E_1 + E_2 : \textbf{float}} \qquad \dfrac{A \vdash E_1 : \textbf{float} \quad A \vdash E_2 : \textbf{int}}{A \vdash E_1 + E_2 : \textbf{float}}$$

While

- For statements that do not have a value, use the type **unit** to represent their result type (**unit** = completed succesfully)

$$\dfrac{A \vdash E : \textbf{bool} \quad A \vdash S : T}{A \vdash \text{while } (E) \ S : \textbf{unit}} \quad \text{(while)}$$

If statements

- Iota: the value of an if statement (if any) is the value of the arm that is executed.
- If no else clause, no value:

$$\dfrac{A \vdash E : \textbf{bool} \quad A \vdash S : T}{A \vdash \text{if } (E) \ S : \textbf{unit}} \quad \text{(if)}$$

Assignment

$$\dfrac{id : T \in A \quad A \vdash E : T}{A \vdash id = E : T} \quad \text{(assign)}$$

$$\dfrac{A \vdash E_3 : T \quad A \vdash E_2 : \text{int} \quad A \vdash E_1 : \text{array}[T]}{A \vdash E_1[E_2] = E_3 : T} \quad \text{(array-assign)}$$

Sequence

- Rule: A sequence of statements is type-safe if the first statement is type-safe, and the remaining are type-safe too:

$$\dfrac{A \vdash S_1 : T_1 \quad A \vdash S_2 ; S_3 ; \ldots ; S_n : T_n}{A \vdash S_1 ; S_2 ; \ldots ; S_n : T_n} \quad \text{(block)}$$

- What about variable declarations ?

Declarations

=unit if no E

$$\frac{A \vdash id : T\ [= E] \quad : T_1 \qquad A \cup \{ id : T \} \vdash S_2 ; \ldots ; S_n : T_n}{A \vdash id : T\ [= E];\ S_2 ; \ldots ; S_n : T_n} \text{ (decl)}$$

- This formally describes the type-checking code from two lectures ago!

Implementation

```
class Block { Stmt stmts[];
   Type typeCheck(SymTab s) { Type t;
     for (int i = 0; i < stmts.length; i++) {
       t = stmts[i].typeCheck(s);
       if (stmts[i] instanceof Decl)
         s = s.add(Decl.id, Decl.typeExpr.evaluate());
     }
     return t;
   }
}
```

$$\frac{A \vdash id : T\ [= E] \quad : T \qquad A \cup \{ id : T \} \vdash S_2; \ldots; S_n : T_n}{A \vdash id : T\ [= E];\ S_2; \ldots ; S_n : T_n} \qquad \frac{A \vdash S_1 : T_1 \quad S_1 \text{ not a decl.} \quad A \vdash S_2 ; S_3; \ldots; S_n : T_n}{A \vdash S_1 ; S_2 ; \ldots ; S_n : T_n}$$

Function application

- If expression E is a function value, it has a type $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- T_i are argument types; T_r is return type
- How to typecheck $E(E_1,\ldots,E_n)$?

$$\frac{A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \qquad A \vdash E_i : T_i \quad (i \in 1..n)}{A \vdash E(E_1,\ldots,E_n) : T_r} \text{ (fnapp)}$$

Function-checking rule

- Iota function syntax

$$id\ (a_1 : T_1,\ldots,a_n : T_n) : T_r = E$$

- Type of *E* must match declared return type of function (E : T), but in what environment?

Add arguments to enviroment!

- Let A be the environment surrounding the function declaration. Function

  $$id\ (a_1 : T_1,\ldots,a_n : T_n) : T_r = E$$

  is type-safe if

  $$A \cup \{ a_1 : T_1, \ldots, a_n : T_n \} \vdash E : T_r$$

- What about recursion?

Example

```
fact(x: int) : int = {
   if (x==0) 1; else x * fact(x - 1); }
```

$$\frac{A2 \vdash fact : int \rightarrow int \qquad \frac{A2 \vdash x : int \quad A2 \vdash 1 : int}{A2 \vdash x\text{-}1 : int}}{}$$

$$\frac{\frac{A2 \vdash x : int \quad A2 \vdash 0 : int}{A2 \vdash x{=}{=}0 : bool} \qquad A2 \vdash 1 : int \qquad \frac{A2 \vdash x : int \quad A2 \vdash fact(x - 1) : int}{A2 \vdash x * fact(x - 1) : int}}{\{fact: int \rightarrow int,\ x : int\} \vdash \text{if } (x{=}{=}0) \ldots;\ \text{else} \ldots : int}$$

How to check return?

$$\frac{A \vdash E : T}{A \vdash \text{return } E : \textbf{none}} \quad \text{(return1)}$$

- A return statement has no value and does not even give control back to enclosing context: special type **none**
- Not the same thing as **unit!**
- But... how do make sure the return type of the current function is T ?

Put it in the symbol table

- Add entry {**return** : int } when we start checking the function, look up this entry when we hit a return statement.
- To check $id\ (a_1 : T_1, \ldots, a_n : T_n) : T_r = E$, in environment A, check

$$A \cup \{ a_1 : T_{1,}, \ldots,\ a_n : T_n\ , \textbf{return} : T \} \vdash\ E : T$$

$$\frac{A \vdash E : T \qquad \text{return} : T \in A}{A \vdash \text{return } E\ :\ \textbf{none}} \quad \text{(return)}$$

Completing static semantics

- Rest of static semantics written in this style
- Provides complete recipe for how to show a program type-safe
- Induction on size of expressions
  - have axioms for atoms: $\overline{A \vdash \text{true} : \textbf{bool}}$
  - for every valid syntactic construct in language, have a rule showing how to prove it type-safe in terms of smaller exprs
- Therefore, have rules for checking all syntactically valid programs for type safety
- & Type checker always terminates!