CS 412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 10: Static Semantics

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

T y p e j u d g m e n t : \ A \ / G \ E : \ T
– means "In the environment A (symbol table), the expression E is a well-typed expression with the 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 a bool (b : bool)
  - 2 must be an int (2 : int)
  - 3 must be an int (3 : int)

Type Judgments

- Type judgment: A ⊢ 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: \text{bool}, x: \text{int}\} \vdash \text{if } (b) \ 2 \ \text{else } x : \text{int}

- Need to show:
  \{b: \text{bool}, x: \text{int}\} \vdash b : \text{bool}
  \{b: \text{bool}, x: \text{int}\} \vdash 2 : \text{int}
  \{b: \text{bool}, x: \text{int}\} \vdash x : \text{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

\[
\begin{array}{c}
\text{Premises} \\
A \vdash E : \text{bool} \\
A \vdash S_1 : T \\
A \vdash S_2 : T
\end{array}
\]
\[
\Rightarrow
\begin{array}{c}
A \vdash \text{if } (E) \ S_1 \ \text{else } S_2 : T
\end{array}
\]

Meaning of Inference Rule

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

\[
\begin{array}{c}
A \vdash E_1 : \text{int} \\
A \vdash E_2 : \text{int}
\end{array}
\]
\[
\Rightarrow
\begin{array}{c}
A \vdash E_1 + E_2 : \text{int}
\end{array}
\]

Implementing a rule

- Work backward from goal:
  \( T = E \text{.typeCheck}(A) \)
  \( \iff \)
  \( A \vdash E : T \)

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 \}

\[
\begin{align*}
A₁ \vdash b: \text{bool} & \quad A₁ \vdash 2: \text{int} & \quad A₁ \vdash 3: \text{int} \\
A₁ \vdash b: \text{bool} & \quad A₁ \vdash 2+3: \text{int} & \quad A₁ \vdash x: \text{int} \\
\{ b: \text{bool}, x: \text{int} \} \vdash \text{if (b) 2+3 else x}: \text{int}
\end{align*}
\]

More about Inference Rules

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

\[
\begin{align*}
A \vdash E_1: \text{float} & \quad A \vdash E_2: \text{float} \\
A \vdash E_1 \vdash: \text{int} & \quad A \vdash E_2 \vdash: \text{int} \\
A \vdash E_1 + E_2: \text{float} & \quad A \vdash E_1 + E_2: \text{float}
\end{align*}
\]

While

- For statements that do not have a value, use the type \text{unit} to represent their result type (\text{unit} = completed successfully)

\[
\begin{align*}
A \vdash E: \text{bool} & \quad A \vdash S: T \\
A \vdash \text{while (E) S}: \text{unit}
\end{align*}
\]

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:

\[
\begin{align*}
A \vdash E: \text{bool} & \quad A \vdash S: T \\
A \vdash \text{if (E) S}: \text{unit}
\end{align*}
\]

Assignment

\[
\begin{align*}
id: T & \in A \\
A \vdash E: T & \quad (assign) \\
A \vdash id = E: T \\
A \vdash E_3: T & \quad (array-assign) \\
A \vdash E_2: \text{int} \\
A \vdash E_1: \text{array}[T] \\
A \vdash E_1[E_2] = E_3: T
\end{align*}
\]

Sequence

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

\[
\begin{align*}
A \vdash S_1: T_1 \\
A \vdash S_2: T_2 \\
\vdots \\
A \vdash S_n: T_n \\
A \vdash S_1; S_2; \ldots; S_n: T_n & \quad (block)
\end{align*}
\]

- What about variable declarations?
Declarations

\[ A \vdash id : T \equiv E : T_i \]

\[ A \cup \{ id : T \} : S_2, ..., S_n : T_n \]  (decl)

\[ A \vdash id : T \equiv E : S_2, ..., S_n : T_n \]

- 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;
    }
}

Function application

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

\[
A \vdash E : T_1 \times T_2 \times ... \times T_n \rightarrow T_r \\
A \vdash E_1 : T_1 \text{ is } i \\
A \vdash E(E_1, ..., E_n) : T_r \\
\]

(\text{fnapp})

Function-checking rule

- Iota function syntax

\[
id (a_1 : T_1, ..., a_n : T_n) : T_r = E
\]

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

Add arguments to environment!

- Let \( A \) be the environment surrounding the function declaration. Function

\[
id (a_1 : T_1, ..., a_n : T_n) : T_r = E
\]

is type-safe if

\[
A \cup \{ a_1 : T_1, ..., a_n : T_n \} \vdash E : T_r
\]

- What about recursion?

Example
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;
    }
}
How to check return?

\[
A \vdash E : T \\
A \vdash \text{return } E : \text{none} \text{ (return1)}
\]

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

\[
A \cup \{a_1 : T_1, \ldots, a_n : T_n, \text{return : } T\} \vdash E : T
\]

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: \( A \vdash \text{true : bool} \)
  - 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!