CS412/413

Introduction to Compilers
Radu Rugina

Lecture 13: Static Semantics
23 Feb 04

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 syntax analysis
  - As regular expression is to lexical analysis
- Static semantics defines types for legal ASTs in the language

Type Judgments

- Static semantics = formal notation which describes type judgments:
  \[ E : T \]
  means “E is a well-typed expression of type T”

- Type judgment examples:
  \[
  \begin{align*}
  2 & : \text{int} \\
  \text{true} & : \text{bool} \\
  "\text{Hello}" & : \text{string}
  \end{align*}
  \]

Type Judgments for Statements

- Statements may be expressions (i.e., represent values)
- Use type judgments for statements:
  \[
  \begin{align*}
  \text{if} (b) \text{ then } 2 \text{ else } 3 & : \text{int} \\
  x & = \text{false} : \text{bool} \\
  b & = \text{true}, y = 2 : \text{int}
  \end{align*}
  \]

- For statements which are not expressions: use a special unit type (empty type):
  \[ S : \text{unit} \]
  means “S is a well-typed statement with no result type”

Deriving a Judgment

- Consider the judgment:
  \[ \text{if} (b) \text{ then } 2 \text{ else } 3 : \text{int} \]

- What do we need to decide that this is a well-typed expression of type \text{int}?
  \[
  \begin{align*}
  b & \text{ must be a bool (b: bool)} \\
  2 & \text{ must be an int (2: int)} \\
  3 & \text{ must be an int (3: int)}
  \end{align*}
  \]

Type Judgments

- Type judgment notation: \[ A \vdash E : T \]
  means “In the context A the expression E is a well-typed expression with the type T”

- Type context is a set of type bindings \[ \text{id} : T \]
  (i.e., type context = symbol table)
  \[
  \begin{align*}
  b & : \text{bool}, x : \text{int} \vdash b : \text{bool} \\
  b, \text{bool}, x : \text{int} & \vdash \text{if} (b) \text{ then } 2 \text{ else } x : \text{int} \\
  & \vdash 2 + 2 : \text{int}
  \end{align*}
  \]
**Deriving a Judgement**

- To show:
  \[ b: \text{bool}, x: \text{int} \vdash (b) \text{ then } 2 \text{ else } x : \text{int} \]
- Need to show:
  \[
  \begin{align*}
  b: \text{bool, x: int} & \vdash b : \text{bool} \\
  b: \text{bool, x: int} & \vdash 2 : \text{int} \\
  b: \text{bool, x: int} & \vdash x : \text{int}
  \end{align*}
  \]

**General Rule**

- For any environment A, expression E, statements S₁ and S₂, the judgment
  \[ A \vdash (E) \text{ then } S₁ \text{ else } S₂ : T \]
  is true if:
  \[
  \begin{align*}
  A & \vdash E : \text{bool} \\
  A & \vdash S₁ : T \\
  A & \vdash S₂ : T
  \end{align*}
  \]

**Inference Rules**

- Premises
  \[ A \vdash E : \text{bool} \quad A \vdash S₁ : T \quad A \vdash S₂ : T \]
- Conclusion
  \[ A \vdash \text{if } (E) \text{ then } S₁ \text{ else } S₂ : T \]
  (if-rule)
- Holds for any choice of E, S₁, S₂, T

**Why Inference Rules?**

- Inference rules: compact, precise language for specifying static semantics (can specify languages in ~20 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

**Meaning of Inference Rule**

- Inference rule says:
  - given that antecedent judgments are true
  - with some substitution for \( A, E₁, E₂ \)
  then, consequent judgment is true
  - with a consistent substitution
  \[
  \begin{align*}
  A & \vdash E₁ : \text{int} \\
  A & \vdash E₂ : \text{int}
  \end{align*}
  \]
  \[
  A \vdash E₁ + E₂ : \text{int} \quad \text{(+) rule}
  \]

**Proof Tree**

- Expression is well-typed if there exists a type derivation for a type judgment
- Type derivation is a proof tree
- Example: if A₁ = b: bool, x: int, then:
  \[
  \begin{align*}
  A₁ & \vdash b : \text{bool} \\
  A₁ & \vdash 2 : \text{int} \\
  A₁ & \vdash 3 : \text{int} \\
  A₁ & \vdash \text{if } (!b) \text{ then } 2+3 \text{ else } x : \text{int}
  \end{align*}
  \]
More about Inference Rules

- No premises = axiom
  \[ A \vdash \text{true : bool} \]
- A goal judgment may be proved in more than one way
  \[
  \begin{align*}
  A \vdash E_1 : \text{float} & \quad A \vdash E_2 : \text{float} \\
  A \vdash E_1 + E_2 : \text{float} & \quad A \vdash E_2 + E_1 : \text{float}
  \end{align*}
  \]
- No need to search for rules to apply -- they correspond to nodes in the AST

While Statements

- Rule for while statements:
  \[
  \begin{align*}
  A \vdash E : \text{bool} & \quad A \vdash S : T \\
  A \vdash \text{while } (E) \ S : \text{unit}
  \end{align*}
  \]
- Why unit type?

If Statements

- If statement as an expression (e.g., in ML): its value is the value of the branch that is executed
  \[
  \begin{align*}
  A \vdash E : \text{bool} & \quad A \vdash S_1 : T \\
  A \vdash S_2 : T \\
  A \vdash \text{if } (E) \ \text{then } S_1 \ \text{else } S_2 : T
  \end{align*}
  \]
- If no else clause:
  \[
  \begin{align*}
  A \vdash E : \text{bool} & \quad A \vdash S : T \\
  A \vdash \text{if } (E) \ S : ?
  \end{align*}
  \]

Assignment Statements

- id : T ∈ A
  \[
  A \vdash E : T
  \]
- (variable-assign)
  \[
  A \vdash \text{id} = E : T
  \]
- id : T
  \[
  A \vdash E_2 : \text{int}
  \]
- (array-assign)
  \[
  A \vdash E_1 : \text{array}[T]
  \]
- A ⊑ E_2 = E_3 : T

Sequence Statements

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:
  \[
  \begin{align*}
  A \vdash S_1 : T_1 & \quad A \vdash (S_2 ; \ldots ; S_n) : T_n \\
  A \vdash (S_1 ; S_2 ; \ldots ; S_n) : T_n
  \end{align*}
  \]
- (sequence)
- What about variable declarations?

Declarations

- A ⊑ id : T [ E ] : T
  \[
  A_0 \vdash \text{id} : T
  \]
- (declaration)
- A ⊑ (id : T [ E ]; S_2 ; \ldots ; S_n) : T

- Declarations add entries to the environment (in the symbol table)
Function Calls

- 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 type-check function call $E(E_1, \ldots, E_n)$?

$$
A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \\
A \vdash E_i : T_i \quad \text{(for i = 1..n)} \\
A \vdash E(E_1, \ldots, E_n) : T_r \\
$$
(function-call)

Function Declarations

- Consider a function declaration of the form $T_r \ \text{fun} \ (T_1 \ a_1, \ldots, \ T_n \ a_n) \ \{ \ \text{return E;} \ \}$
- Type of function body $S$ must match declared return type of function, i.e. $E : T_r$
- ... but in what type context?

Add Arguments to Environment!

- Let $A$ be the context surrounding the function declaration. Function declaration:

$$
T_r \ \text{fun} \ (T_1 \ a_1, \ldots, \ T_n \ a_n) \ \{ \ \text{return E;} \ \}
$$

is well-formed if

$$
A, a_i : T_i, \ldots, a_n : T_n \vdash E : T_r
$$

...what about recursion?

Need: $\text{fun:} \ T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \in A$

Recursive Function Example

- Factorial:

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

- Prove: $A \vdash x \cdot \text{fact}(x-1) : \text{int}$
  Where: $A = (\text{fact: int} \rightarrow \text{int}, x : \text{int})$

Mutual Recursion

- Example:

```c
int f(int x) { return g(x) + 1; }
int g(int x) { return f(x) - 1; }
```

- Need environment containing at least

$$
f : \text{int} \rightarrow \text{int}, g : \text{int} \rightarrow \text{int}
$$

when checking both $f$ and $g$

- Two-pass approach:
  - Scan top level of AST picking up all function signatures and creating an environment binding all global identifiers
  - Type-check each function individually using this global environment

How to Check Return?

$$
A \vdash E : T \\
A \vdash \text{return} E : \text{unit}
$$

- A return statement produces no value for its containing context to use
- Does not return control to containing context
- Suppose we use type unit...
- ...then how to make sure the return type of the current function is $T$?
**Put Return in the Symbol Table**

- Add a special entry `{ return_fun : T }` when we start checking the function "fun", look up this entry when we hit a return statement.
- To check $T$, `fun (T_1, a_1, ..., T_n, a_n) { return S; }` in environment $A$, need to check:

  $$
  A, a_i : T_i, ..., a_n : T_n, \text{return}_\text{fun} : T_j \vdash S : T_j
  $$

  $$
  \begin{align*}
  A \vdash E : T & \quad \text{return}_\text{fun} : T \in A \\
  A \vdash \text{return} E : \text{unit}
  \end{align*}
  $$

**Static Semantics Summary**

- **Static semantics** = formal specification of type-checking rules
- Concise form of static semantics: typing rules expressed as inference rules
- Expression and statements are well-formed (or well-typed) if a typing derivation (proof tree) can be constructed using the inference rules