CS412/413
Introduction to Compilers
Radu Rugina

Lecture 13: Static Semantics
18 Feb 02

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:
  2 : int
  true : bool
  "Hello" : string

Type Judgments for Statements
• Statements may be expressions (i.e. represent values)
• Use type judgments for statements:
  if (b) 2 else 3 : int
  x = 10 : bool
  b = true, y = 2 : int
• For statements which are not expressions: use a special unit type (void or empty type):
  S : unit
  means "S is a well-typed statement with no result type"

Deriving a Judgment
• Consider the 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 notation: A ⊢ 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 id : T
  (i.e. type context = symbol table)
  b : bool, x : int ⊢ b : bool
  b : bool, x : int ⊢ if (b) 2 else x : int
  ⊢ 2 + 2 : int
Deriving a Judgement

• To show:
  \[ b : \text{bool}, x : \text{int} \vdash (b \land \text{if (b) 2 else } x) : \text{int} \]

• Need to show:
  \[ b : \text{bool}, x : \text{int} \vdash b : \text{bool} \]
  \[ b : \text{bool}, x : \text{int} \vdash \neg b : \text{int} \]
  \[ b : \text{bool}, x : \text{int} \vdash x : \text{int} \]

General Rule

• For any environment \( A \), expression \( E \), statements \( S_1 \) and \( S_2 \), the judgment
  \[ A \vdash (E \land \text{if (E) } S_1 \text{ else } S_2) : T \]
  is true if:
  \[ A \vdash E : \text{bool} \]
  \[ A \vdash S_1 : T \]
  \[ A \vdash S_2 : T \]

Inference Rules

Premises

\[ A \vdash E : \text{bool} \]
\[ A \vdash S_1 : T \]
\[ A \vdash S_2 : T \]

(\( \land \)-rule)

Conclusion

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

Why Inference Rules?

• Inference rules: compact, precise language for specifying static semantics (can specify languages in \( \sim 20 \) 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 \( A, E_1, E_2 \)
  - then, consequent judgment is true
  - with a consistent substitution

\[ A \vdash E_1 : \text{int} \]
\[ A \vdash E_2 : \text{int} \]

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

Proof Tree

• Expression is well-typed if there exists a type derivation for a type judgment
• Type derivation is a proof tree
• Example: if \( A_1 = b : \text{bool}, x : \text{int}, \text{then}:

\[ A_1 \vdash b : \text{bool} \]
\[ A_1 \vdash x : \text{int} \]

\[ A_1 \vdash \text{if (b) } 2 + 3 \text{ else } x : \text{int} \]
More about Inference Rules

- No premises = axiom
  \[ A \vdash \text{true} : \text{bool} \]

- A goal judgment may be proved in more than one way
  \[
  \begin{align*}
  A &\vdash \text{E}_1 : \text{float} \\
  A &\vdash \text{E}_2 : \text{float} \\
  A &\vdash \text{E}_1 + \text{E}_2 : \text{float}
  \end{align*}
  \]

- No need to search for rules to apply -- they correspond to nodes in the AST

While Statements

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

- Why use unit type for while statements?

If Statements

- If statement as an expression: its value is the value of the branch that is executed
  \[
  \begin{align*}
  A &\vdash \text{E} : \text{bool} \\
  A &\vdash \text{S}_1 : \text{T} \\
  A &\vdash \text{S}_2 : \text{T} \\
  A &\vdash \text{if (E) S}_1 \text{ else } S_2 : \text{T}
  \end{align*}
  \]

- If no else clause, no value (why?)
  \[
  \begin{align*}
  A &\vdash \text{E} : \text{bool} \\
  A &\vdash \text{S} : \text{T} \\
  A &\vdash \text{if (E) S} : \text{unit}
  \end{align*}
  \]

Assignment Statements

- \[ \text{id : T} \in A \]
  \[ A \vdash \text{E} : \text{T} \]
  \[ A \vdash \text{id = E} : \text{T} \]
  \[ \text{(variable-assign)} \]

- \[ A \vdash \text{E}_2 : \text{T} \]
  \[ A \vdash \text{E}_2 : \text{int} \]
  \[ A \vdash \text{E}_1 : \text{array[T]} \]
  \[ A \vdash \text{E}_1[E_2] = \text{E}_2 : \text{T} \]
  \[ \text{(array-assign)} \]

Sequence Statements

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed too:
  \[
  \begin{align*}
  A &\vdash \text{S}_1 : \text{T}_1 \\
  A &\vdash \text{(S}_2 ; \ldots ; \text{S}_n) : \text{T}_n
  \end{align*}
  \]

- What about variable declarations?

Declarations

- \[ A \vdash \text{id : T} = \text{E} : \text{T}_1 \]
  \[ A, v \vdash \text{(id : T} = \text{E} ; \text{S}_2 ; \ldots ; \text{S}_n) : \text{T}_n \]
  \[ \text{(declaration)} \]

- 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 \( E(E_{i_1}, \ldots, E_{i_n}) \)?

\[
A \leftarrow E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \\
A \leftarrow E_i : T_i \quad (i \in \{1,2,\ldots\}) \\
A \leftarrow E(E_{i_1}, \ldots, E_{i_n}) : T_r \\
\text{(function-call)}
\]

Function Declarations

- Consider a function declaration of the form
  \( T_r \ \text{fun} \ (T_1 \ a_1, \ldots, T_n \ a_n) = E \)
  (equivalent to: \( T_r \ \text{fun} \ (T_1 \ a_1, \ldots, T_n \ a_n) \ {\text{return}} \ E; \) )
- 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:
  \( T_r \ \text{fun} \ (T_1 \ a_1, \ldots, T_n \ a_n) = E \)
  is well-formed if
  \( A, a_1 : T_1; \ldots; a_n : T_n \rightarrow E : T_r \)
- ...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:
  \[
  \text{int fact(int } x) = \\
  \quad \text{if } (x == 0) 1; \text{ else } x \times \text{ fact}(x - 1);
  \]
- Prove: \( A \leftarrow (x == 0) \ldots \text{ else } \ldots ; \text{ int} \)
  Where: \( A = \{ \text{fact: int} \rightarrow \text{int, } x : \text{int} \} \)

Mutual Recursion

- Example:
  \[
  \text{int f(int } x) = g(x) + 1; \\
  \text{int g(int } x) = f(x) - 1;
  \]
- 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?

\[
A \leftarrow \text{E : T} \quad \text{(return)} \\
A \leftarrow \text{return } E : \text{unit}
\]

- 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 \{ \text{return, fun} : T \} when we start checking the function "fun", look up this entry when we hit a return statement.
- To check \( T_f \text{ fun} (T_1, a_1, \ldots, a_n, \text{return, fun}) \ (S) \) in environment \( A \), need to check:

\[
A, a_1 : T_1, \ldots, a_n : T_n, \text{return, fun} : T_f \vdash S : T_f
\]

\[
\begin{align*}
A \vdash E : T & \quad \text{return, fun} : T \in A \\
A \vdash \text{return} : \text{unit}
\end{align*}
\]  

(\text{return})

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