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　　2 * (3 + 4) : 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: $A \vdash E : T$

  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 id : T

  (i.e. type context = symbol table)

  b: bool, x: int $\vdash$ b : bool
  b: bool, x: int $\vdash$ if (b) 2 else x : int
  $\vdash$ 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 ⊢ if (E) S_1 else S_2 : T

  is true if:

  A ⊢ E : bool

  A ⊢ S_1 : T

  A ⊢ S_2 : T

Inference Rules

Premises

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

Conclusion

- Holds for any choice of E, S_1, S_2, 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 ⊢ E : T true by walking backward through rules

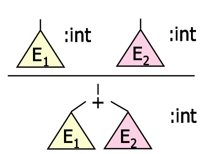
Meaning of Inference Rule

- 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

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



Proof Tree

- Expression is well-typed if there exists a type derivation for a type judgment
- Type derivation is a proof tree
- Example: if A1 = b: bool, x: int, then:

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

More about Inference Rules

- No premises = axiom

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

- A goal judgment may be proved in more than one way

$$\frac{A \vdash E_1 : \text{float} \quad A \vdash E_2 : \text{float}}{A \vdash E_1 + E_2 : \text{float}} \qquad \frac{A \vdash E_1 : \text{float} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{float}}$$

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

While Statements

- Rule for while statements:

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

- 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

$$\frac{A \vdash E : \text{bool} \quad A \vdash S_1 : T \quad A \vdash S_2 : T}{A \vdash \text{if } (E)\ S_1 \text{ else } S_2 : T}\ (\text{if-then-else})$$

- If no else clause, no value (why?)

$$\frac{A \vdash E : \text{bool} \quad A \vdash S : T}{A \vdash \text{if } (E)\ S : \text{unit}}\ (\text{if-then})$$

Assignment Statements

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

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

Sequence Statements

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

$$\frac{A \vdash S_1 : T_1 \quad A \vdash (S_2 ; \ldots ; S_n) : T_n}{A \vdash (S_1 ; S_2 ; \ldots ; S_n) : T_n}\ (\text{sequence})$$

- What about variable declarations ?

Declarations

= unit if no E

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

- 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 $E(E_1,\ldots,E_n)$?

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

Function Declarations

- Consider a function declaration of the form

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

  (equivalent to: T_r fun $(T_1\ a_1,\ldots, T_n\ a_n)$ { 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 fun $(T_1\ a_1,\ldots, T_n\ a_n)$ = E

  is well-formed if

  $A, a_1 : T_1, \ldots, a_n : T_n \vdash E : T_r$
- …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:

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

- Prove: $A \vdash$ if (x==0) … else … : int

  Where: $A = \{ \text{fact}: \text{int} \rightarrow \text{int}, x : \text{int} \}$

Mutual Recursion

- Example:

```
int f(int x) = g(x) + 1;
int g(int x) = f(x) – 1;
```

- Need environment containing at least

  f: int → int, g: int → 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?

$$\frac{A \vdash E : T}{A \vdash \text{return } E : \text{unit}} \text{ (return1)}$$

- 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_r fun (T_1 a_1,…, T_n a_n) { S } in environment A, need to check:

$$A, a_1 : T_1, \ldots, a_n : T_n, \text{return\_fun} : T_r \vdash S : T_r$$

$$\frac{A \vdash E : T \quad \text{return\_fun} : T \in A}{A \vdash \text{return } E : \text{unit}} \text{(return)}$$

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