CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 13: Static Semantics

18 Feb 08

Type Inference Systems

- Type inference systems are a declarative formal system used to define typings for legal programs in a language

- Type inference systems are to type-checking:
  - As regular expressions are to lexical analysis
  - As context-free grammars are to syntax analysis

- Type inference systems are examples of the more general notion: natural semantics

Type Judgments

- The type judgment:

  $\vdash E : T$

  is read:

  "E is a well-typed construct of type T"

- Type checking program P is demonstrating the validity of the type judgment $\vdash P : T$ for some type T

- Sample valid type judgments for program fragments:

  $\vdash 2 : int$ $\qquad\qquad$ $\vdash 2 * (3 + 4) : int$

  $\vdash true : bool$ $\qquad\qquad$ $\vdash (true\ ?\ 2 : 3) : int$

Deriving a Type Judgment

- Consider the judgment:

  |– (b ? 2 : 3) : int

- What do we need in order to decide that this is a valid type judgment?

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

Hypothetical Type Judgments

- The hypothetical type judgment

  A |– E : T

  is read:

  "In type context A expression E is well-typed with type T"

- A type context is a mapping of identifiers to types (i.e., a symbol table)

- Sample valid hypothetical type judgments:

  b: bool |– b : bool

  |– 2 + 2 : int

  b: bool, x: int |– (b ? 2 : x) : int

  b: bool, x: int |– b : bool

  b: bool, x: int |– 2 + 2 : int

- Type checking program P is demonstrating the validity of A|– P : T for some type T and the language's standard environment A

Deriving a Type Judgment

- To show:

  b: bool, x: int |– (b ? 2 : 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 type environment A, expressions E, E_1 and E_2, the judgment

$$A \mid\!- (E \; ? \; E_1 : E_2) : T$$

is valid if:

$$A \mid\!- E : \text{bool}$$
$$A \mid\!- E_1 : T$$
$$A \mid\!- E_2 : T$$

Inference Rule Schema

$$\overbrace{\dfrac{A \vdash E: \text{bool} \qquad A \vdash E_1: T \qquad A \vdash E_2: T}{\underbrace{A \vdash (E \; ? \; E_1 : E_2) : T}_{\text{Conclusion (a.k.a., consequent)}}}}^{\text{Premises (a.k.a., antecedant)}} \quad \text{(if-rule)}$$

- Holds for any choice of A, E, E_1, E_2, and T
- An inference rule schema defines an infinite number of inference rules

Axioms

- An axiom is an inference rule (schema) with no premises

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

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 are to type inference systems as productions are to context-free grammars
- Type judgments are to type inference systems as nonterminals are to context-free grammars
- Type checking is an attempt to prove that a type judgment is A |– E : T is valid

Meaning of Inference Rule

- Inference rule says:

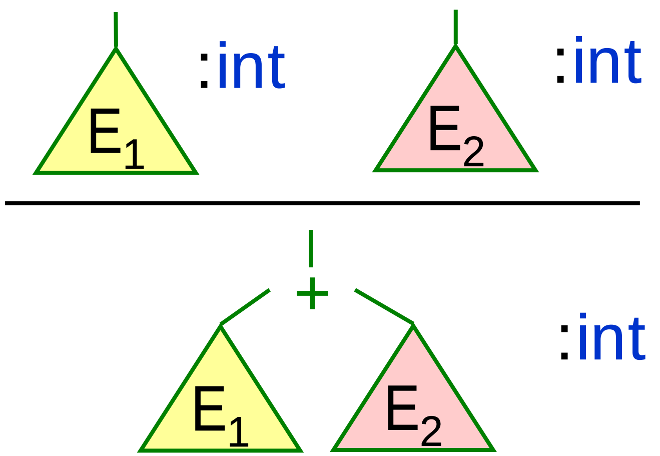
given that the antecedent judgments are derivable

  – with a uniform substitution for meta-variables (i.e., A, E_1, E_2)

then the consequent judgment is derivable

  – with the same uniform substitution for the meta-variables

$$\frac{\begin{array}{c} A \vdash E_1 : \text{int} \\ A \vdash E_2 : \text{int} \end{array}}{A \vdash E_1 + E_2 : \text{int}}\ (+)$$



Proof Tree

- A construct is well-typed if there exists a type derivation for a type judgment for the construct
- Type derivation is a proof tree where all the leaves are axioms
- Example: if A1 = b: bool, x: int, then:

$$\dfrac{\dfrac{\dfrac{}{A1 \;|\!- \; b : \text{bool}}}{A1 \;|\!- \; !b : \text{bool}} \qquad \dfrac{\dfrac{}{A1 \;|\!- \; 2 : \text{int}} \quad \dfrac{}{A1 \;|\!- \; 3 : \text{int}}}{A1 \;|\!- \; 2+3 : \text{int}} \qquad \dfrac{}{A1 \;|\!- \; x : \text{int}}}{A1 \;|\!- \; (\; !b \;?\; 2+3 : x \;) : \text{int}}$$

Proof Tree, cont.

- Axioms are analogous to production with epsilon on the right hand side
- A complete proof of A |– E : T is like a derivation of epsilon from A |– E : T

Type Judgments for Statements

- Statements that have no value are said to have type void, i.e., judgment

  $\vdash S : \text{void}$

  means "S is a well-typed statement with no result type"

- ML uses unit instead of void

While Statements

- Rule for while statements:

$$\frac{\begin{array}{c} A \;\vdash E : \text{bool} \\ A \;\vdash S : T \end{array}}{A \;\vdash \textbf{while}\ (E)\ S : \text{void}} \;\; \text{(while)}$$

Assignment (Expression) Statements

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

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

Sequence Statements

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

$$\frac{\begin{array}{c} A \mid\!\!- S_1 : T_1 \\ A \mid\!\!- (S_2 \,;\, \ldots \,;\, S_n) : T_n \end{array}}{A \mid\!\!- (S_1 \,;\, S_2 \,;\, \ldots \,;\, S_n) : T_n} \quad \text{(sequence)}$$

Identifier Declaration List

- What about variable declarations (with initialization)?
- Declarations add entries to the type environment in which the scope of the declared variable must type check

$$\frac{A \vdash E : T \qquad A, id : T \vdash (S_2 \,;\, \ldots \,;\, S_n) \;:\; T'}{A \vdash (id : T = E \,;\, S_2 \,;\, \ldots \,;\, S_n) \;:\; T'} \quad \text{(declaration)}$$

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

  $T_r\ \ f\ (T_1\ a_1, \ldots, T_n\ a_n)\ \ \{\ E;\ \}$

- The body of the function must type check in an environment containing the type bindings for the formal parameters

$$\frac{A,\ a_1 : T_1, \ldots,\ a_n : T_n \ \vdash E : T_r}{A \vdash T_r\ \ f\ (T_1\ a_1, \ldots, T_n\ a_n)\ \ \{\ E;\ \} : \text{void}}\ \text{(function-body)}$$

But what about recursion?

- Example:

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

- Need to prove: $A \vdash x * fact(x-1) : int$

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

And mutual recursion?

- Example:

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

- Need environment containing at least

  $f: int \rightarrow int,\ g: int \rightarrow int$

  when checking both f and g

- Two-pass approach needed:
  - First pass: collect all function signatures into a type environment A
  - Second pass: type-check each function declaration using this global environment A
  - How to express this with type inference schema is left as an exercise

How to Check Return?

$$\frac{A \vdash E : T}{A \vdash \textbf{return}\ E : \text{void}} \quad \text{(return1)}$$

- A return statement produces no value for its containing context to use
- Does not return control to containing context

- Suppose we use type void...
- ...then how to make sure T is the return type of the current function?

Put return type in environment

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

$$\frac{A,\ a_1 : T_1, \ldots,\ a_n : T_n,\ return\_f : T_r\ \vdash E : T_r}{A\ \vdash T_r\ \ f\ (T_1\ a_1, \ldots, T_n\ a_n)\ \ \{\ E;\ \}\ :\ void}\ \text{(function-body)}$$

$$\frac{A,\ return\_f : T\ \vdash E : T}{A,\ return\_f : T\ \vdash \textbf{return}\ E : void}\ \text{(return)}$$

Static Semantics Summary

- Type inference system = formal specification of typing 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