CS412/CS413

Introduction to Compilers
Tim Teitelbaum

Lecture 14: Static Semantics
21 Feb 07

Type Inference Systems

- Type inference systems define types for all 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 Judgments

- The type judgment:
  |– E : T
  means:
  "E is a well-typed construct of type T"
- Type judgments are to type inference systems as sentential forms are to context-free grammars
- Type checking program P is demonstrating the validity of the type judgment |– P : T for some type T
- Sample valid type judgments for program fragments:

|– 2 : int |– 2 * (3 + 4) : int
|– true : bool |– (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
  means
  "In the type context A expression E is a well-typed expression with type T "

- A type context is a set of type bindings id : T
  (i.e., a type context is 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

Deriving a 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

$$A \vdash (E \,?\, E_1 : E_2) : T$$

is valid if:

$$A \vdash E : \text{bool}$$
$$A \vdash E_1 : T$$
$$A \vdash E_2 : T$$

Inference Rule Schema

Premises (a.k.a., antecedant)

$$\frac{A \vdash E : \text{bool} \qquad A \vdash E_1 : T \qquad A \vdash E_2 : T}{A \vdash (E \,?\, E_1 : E_2) : T} \ \text{(if-rule)}$$

Conclusion (a.k.a., consequent)

- Holds for any choice of A, E, E_1, E_2, and T

Why Inference Rules?

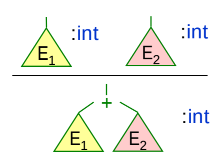
- Inference rules: compact, precise language for specifying static semantics (can specify languages in ~20 pages vs. 100's of pages of Java Language Specification)
- Inference rules are to type inference systems as productions are to context-free grammars
- Type judgments are to type inference systems as non-terminals are to context-free grammars
- Inference rules correspond directly to recursive AST traversal that implements them
- Type checking is attempt to prove that type judgments A |– E : T are derivable

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

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



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
- Type derivations are to type inference systems as derivations are to context-free grammars
- Example: if A1 = b: bool, x: int, then:

$$\dfrac{\dfrac{\dfrac{}{A1 \vdash b : \text{bool}}}{A1 \vdash\, !b : \text{bool}} \quad \dfrac{\dfrac{}{A1 \vdash 2 : \text{int}} \quad \dfrac{}{A1 \vdash 3 : \text{int}}}{A1 \vdash 2+3 : \text{int}} \quad \dfrac{}{A1 \vdash x : \text{int}}}{A1 \vdash (\,!b \,?\, 2+3 : x\,) : \text{int}}$$

More about Inference Rules

- No premises = axiom

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

- An inference rule with no premises is analogous to a production with no non-terminals on the right hand side
- A 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

Type Judgments for Statements

- Statements that have no value are said to have type void, i.e., judgment
  $\vdash S : void$
  means "S is a well-typed statement with no result type"
- ML uses unit instead of void

While Statements

- Rule for while statements:

$$\frac{A \vdash E : bool \qquad A \vdash S : T}{A \vdash \textbf{while}\ (E)\ S : void} \ \text{(while)}$$

- Why void type?

Assignment (Expression) Statements

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

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

Sequence Statements

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

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

Declaration List

- What about variable declarations?
- Declarations add entries to the environment (in the symbol table)

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

Function Calls

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

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

Function Declarations

- Consider a function declaration of the form

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

- Type of function body 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. Then the function declaration

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

  is well-formed if

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

- ...but what about recursion?

  Need: $f: T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \in A$

Recursive Function Example

- Factorial:

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

- Prove: $A \vdash x * fact(x-1) : int$

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

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:
  - 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?

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

- 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 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 "f", look up this entry when we hit a return statement.
- To check $T_r\ f\ (T_1\ a_1, \ldots, T_n\ a_n)$ { **return** S; } in environment A, need to check:

$$A, a_1 : T_1, \ldots, a_n : T_n, return\_f : T_r \vdash S : T_r$$

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

Static Semantics Summary

- Type inference system = 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