CS412/413

Introduction to Compilers
Radu Rugina

Lecture 13 : AST Visitors, Typing Rules
20 Feb 06

AST Traversals

- First construct the AST
- Then traverse the AST to perform semantic checks or other actions
  - At this point tree has been built and is stable
- Possible ways of implementing traversals:
  - Dedicated methods: not extensible
  - `instanceof` + typecasts: error-prone
  - the visitor pattern

    recommended book (the "Gang of Four" book): "Design Patterns", by Gamma, Helm, Johnson, Vlissides

Visitor Methodology for AST Traversal

- Visitor pattern: useful OO programming pattern that separates data structure definition (e.g., the AST) from code that traverses the structure (e.g., the name resolution code and the type checking code).
- Visitor recipe:
  - Define a Visitor interface for all traversals of the AST
  - Extend each AST class with a method that accepts any Visitor
  - Code each traversal as a separate class that implements the Visitor interface

AST Data Structure

```
abstract class Expr { …  }
class Add extends Expr { ...
   Expr e1, e2;
}
class Num extends Expr { ...
   int value;
}
class Id extends Expr { ...
   Symbol id;
}
```


Visitor Interface

```
interface Visitor {
 void visit(Add e);
 void visit(Num e);
 void visit(Id e);
}
```


Accept methods

```
abstract class Expr { …
   abstract public void accept(Visitor v);
}
class Add extends Expr { …
   public void accept(Visitor v) {
      v.visit(this); }
}
class Num extends Expr { …
   public void accept(Visitor v) {
      v.visit(this); }
}
class Id extends Expr { …
   public void accept(Visitor v) {
      v.visit(this); }
}
```

The declared type of **this** is the subclass it which it occurs.

Overload resolution of v.visit(**this**); invokes appropriate visit function in the Visitor.

Visitor Methods

- For each kind of traversal, implement the Visitor interface, e.g.,

```
class PostfixPrintVisitor implements Visitor {
    void visit(Add e)  {
       e.e1.accept(this); e.e2.accept(this);
       System.out.print( "+" );
    }
    void visit(Num e)   {
       System.out.print(value);
    }
    void visit(Id e)   {
       System.out.print(id);
    }
}
```

Dispatch in the visit methods eliminates case analysis on AST subclasses

- To traverse expression e:

```
Visitor v = new PostfixPrintVisitor();
e.accept(v);
```


Inherited and Synthesized Information

- So far, OK for traversal and action w/o communication of values
- But we need a way to pass information
  - Down the AST (called "inherited attributes")
  - Up the AST (called "synthesized attributes")
- To pass information down the AST
  - add parameter to visit functions
- To pass information up the AST
  - add return value to visit functions

Visitor Interface (2)

```
interface Visitor {
 Object visit(Add e, Object inh);
 Object visit(Num e, Object inh);
 Object visit(Id e, Object inh);
}
```


Accept methods (2)

```
abstract class Expr { …
   abstract public Object accept(Visitor v, Object inh);
}
class Add extends Expr { …
   public Object accept(Visitor v, Object inh) {
     return v.visit(this, inh); }
}
class Num extends Expr { …
   public Object accept(Visitor v, Object inh) {
       return v.visit(this, inh);
}
class Id extends Expr { …
   public Object accept(Visitor v, Object inh) {
       return v.visit(this, inh); }
}
```


Visitor Methods (2)

- For kind of traversal, implement the Visitor interface, e.g.,

```
class EvaluationVisitor implements Visitor {
    Object visit(Add e, Object inh)  {
       int left = (int) e.e1.accept(this, inh);
       int right = (int) e.e2.accept(this, inh);
       return left + right;
    }
    Object visit(Num e, Object inh)   {
       return value;
    }
    Object visit(Id e, Object inh)   {
       return Lookup(id, (Environment)inh);
    }
}
```

- To traverse expression e:

```
Visitor v = new EvaluationVisitor ();
e.accept(v, env);
```


Typing Rules

- 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 AST nodes 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:

  `(b ? 2 : 3)` : int

  `x = false` : bool

  `b = true, y = 2` : int

- For statements which are not expressions: use a special void type (empty type); S : void means "S is a well-typed statement with no result type"
- Languages such as ML use a unit type

Deriving a Judgment

- Consider the judgment:

  `(b ? 2 : 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 $\vdash$ `(b ? 2 : x)` : int

  $\vdash$ `2 + 2` : int

Deriving a Judgement

- To show:

  b: bool, x: int $\vdash$ `(b ? 2 : x)` : int

- Need to show:

  b: bool, x: int $\vdash$ b : bool

  b: bool, x: int $\vdash$ 2 : int

  b: bool, x: int $\vdash$ x : int

General Rule

- For any environment A, expression E, statements S_1 and S_2, the judgment

  $$A \vdash (E_1 \mathbin{?} E_2 : E_3) : T$$

  is true if:

  $A \vdash E_1$: bool

  $A \vdash E_2 : T$

  $A \vdash E_3 : T$

Inference Rules

Premises

$$\frac{A \vdash E_1 : \text{bool} \quad A \vdash E_2 : T \quad A \vdash E_3 : T}{A \vdash (E_1 \ ?\ E_2 : E_3) : T} \ (\text{cond})$$

Conclusion

- Holds for any choice of A, E_1, E_1, E_3, 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

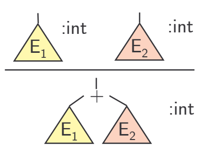
Meaning of Inference Rule

- Inference rule says:

  given that antecedent judgments are true
  - with some substitution for A, 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 A = b: bool, x: int, then:

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

More about Inference Rules

- No premises = axiom

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

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

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

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

While Statements

- All statements have type void
- Judgments of the form: $A \vdash S$
  - "In environment A, statement S is well-typed"
- Rule for while statements:

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

Assignment Statements

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

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

Sequence Statements

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

$$\frac{A \vdash S_1 \quad A \vdash (S_2 \,;\, \dots \,;\, S_n)}{A \vdash (S_1 \,;\, S_2 \,;\, \dots \,;\, S_n)} \text{ (sequence)}$$

- What about variable declarations ?

Declarations

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

- Declarations add entries to the environment (in the symbol table)
- Corresponds to adding `id` to the symbol table

Function/Method Calls

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

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

Function Declarations

- Consider a function declaration of the form

  T_r fun (T_1 a_1,…, 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,…, T_n a_n) { return E; }

  is well-formed if

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

- …what about recursion?

  Need: fun: $T_1 \times T_2 \times \dots \times T_n \rightarrow T_r \in A$

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→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 → int, g: int → int

  when checking both f and g

- Two-pass approach:
  - Parse, build AST and symbol tables
  - Then type-check AST using the information in the symbol tables

How to Check Return?

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

- A return statement produces no value for its containing context to use
- How to make sure the return type of the current function is T ?

Put Return in the Symbol Table

- Add a special entry { `ret` : T_r } 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) { `return` S; } in environment A, need to check:

$$A,\ a_1 : T_1, \ldots,\ a_n : T_n,\ \texttt{ret} : T_r \vdash S : T_r$$

$$\frac{A \vdash E : T \qquad \texttt{ret} : T \in A}{A \vdash \texttt{return}\ E} \quad \text{(return)}$$

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