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

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

Visitor Interface

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

Accept methods

```java
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.,

```java
class PrintExpressionVisitor implements Visitor {
   void visit(Add a) {
      System.out.println("+" + a.left.accept(this) + a.right.accept(this));
   }
   public void visit(Id id) {
      System.out.println(id.name);
   }
}
```

- To traverse expression `e`:

```java
Visitor v = new PrintExpressionVisitor();
v.visit(e);
```

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)

```java
interface Visitor {
   void visit(Add a, Object inh);
   void visit(Id id, Object inh);
}
```

Accept methods (2)

```java
abstract class Expr { ...
   public void accept(Visitor v, Object inh) {
      return v.visit(this, inh);
   }
}
```

Visitor Methods (2)

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

```java
class EvaluationVisitor implements Visitor {
   void visit(Add a, Object inh) {
      int left = (int) a.left.accept(this, inh);
      int right = (int) a.right.accept(this, inh);
      return left + right;
   }
   void visit(Id id, Object inh) {
      return new int[]{id.name};
   }
}
```

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>2 * (3 + 4)</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>true</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>&quot;Hello&quot;</td>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>

Type Judgments for Statements

• Statements may be expressions (i.e. represent values)
• Use type judgments for statements:

\[
\begin{align*}
(b \ ? \ 2 : 3) & : \text{int} \\
x & = \text{false} : \text{bool} \\
b & = \text{true} : \text{int}
\end{align*}
\]

• For statements which are not expressions: use a special void type (empty type): \( S : \text{void} \) 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:

\[ (b \ ? \ 2 : 3) : \text{int} \]

• 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: \( A \vdash E : T \)
• 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)

\[
\begin{align*}
b & : \text{bool} \\
x & : \text{int} \vdash b : \text{bool} \\
b & : \text{bool} \\
x & : \text{int} \vdash (b \ ? \ 2 : x) : \text{int} \\
\vdash 2 + 2 : \text{int}
\end{align*}
\]

Deriving a Judgement

• To show:

\[ b : \text{bool}, x : \text{int} \vdash (b \ ? \ 2 : x) : \text{int} \]

• Need to show:

\[
\begin{align*}
b & : \text{bool} \\
x & : \text{int} \vdash b : \text{bool} \\
b & : \text{bool} \\
x & : \text{int} \vdash 2 : \text{int} \\
b & : \text{bool} \\
x & : \text{int} \vdash x : \text{int}
\end{align*}
\]

General Rule

• For any environment \( A \), expression \( E \), statements \( S_1 \) and \( S_2 \), the judgment

\[ A \vdash (E_1 \ ? \ E_2 : E_3) : T \]

is true if:

\[
\begin{align*}
A & \vdash E_1 : \text{bool} \\
A & \vdash E_2 : T \\
A & \vdash E_3 : T
\end{align*}
\]
Inference Rules

Premises

\[
A \vdash E_1 \colon \text{bool} \quad A \vdash E_2 \colon T \quad A \vdash E_3 \colon T \\
\quad \text{(cond)}
\]

Conclusion

\[
A \vdash (E_1 \ ? \ E_2 \ : E_3) \colon T
\]

• Holds for any choice of \(A, E_1, E_2, E_3, T\)

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 \(A \vdash E \colon T\) 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

\[
\begin{align*}
A & \vdash E_1 \colon \text{int} \\
A & \vdash E_2 \colon \text{int} \\
A & \vdash E_1 + E_2 \colon \text{int} \\
\end{align*}
\]

Proof Tree

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

\[
\begin{align*}
A & \vdash b \colon \text{bool} \\
A & \vdash 2 \colon \text{int} \\
A & \vdash 3 \colon \text{int} \\
A & \vdash x \colon \text{int} \\
A & \vdash 1b \colon \text{bool} \\
A & \vdash 2+3 \colon \text{int} \\
A & \vdash x \colon \text{int} \\
A & \vdash 1b \ ? \ 2+3 \ : x \colon \text{int} \\
\end{align*}
\]

More about Inference Rules

• No premises = axiom

\[
A \vdash \text{true} \colon \text{bool}
\]

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

\[
\begin{align*}
A & \vdash E_1 \colon \text{float} \\
A & \vdash E_2 \colon \text{float} \\
A & \vdash E_1 + E_2 \colon \text{float} \\
\end{align*}
\]

\[
\begin{align*}
A & \vdash E_1 \colon \text{float} \\
A & \vdash E_2 \colon \text{int} \\
A & \vdash E_1 + E_2 \colon \text{float} \\
\end{align*}
\]

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

While Statements

• All statements have type \text{void}
• Judgments of the form: \(A \vdash S\)
  – “In environment \(A\), statement \(S\) is well-typed”

• Rule for while statements:

\[
\begin{align*}
A & \vdash E \colon \text{bool} \\
A & \vdash S \\
A & \vdash \text{while}(E)S \\
\end{align*}
\]
Assignment Statements

\[
\begin{align*}
\text{id} &: T \in A \\
A &\vdash E : T \\
A &\vdash \text{id} = E \quad \text{(variable-assign)} \\
A &\vdash E_3 : T \\
A &\vdash E_2 : \text{int} \\
A &\vdash E_1 : \text{array}(T) \\
A &\vdash E_1[E_2] = E_3 \quad \text{(array-assign)}
\end{align*}
\]

Sequence Statements

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

\[
\begin{align*}
A &\vdash S_1 \\
A &\vdash (S_2 : \ldots : S_n) \\
A &\vdash (S_1 ; S_2 ; \ldots ; S_n) \quad \text{(sequence)}
\end{align*}
\]

- What about variable declarations?

Declarations

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

- Declarations add entries to the environment (in the symbol table)
- Corresponds to adding \text{id} to the symbol table

Function/Method Calls

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

\[
\begin{align*}
A &\vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \\
A &\vdash E_1 : T_i \quad (i = 1 \ldots n) \\
A &\vdash E(E_1, \ldots, E_n) : T_r \quad \text{(function-call)}
\end{align*}
\]

Function Declarations

- Consider a function declaration of the form

\[
\text{} \quad \text{T}_r \text{ fun } (T_1 \ a_1, \ldots, T_n \ a_n) \{ \text{return } E; \}
\]

- 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:

\[
\text{T}_r \text{ fun } (T_1 \ a_1, \ldots, T_n \ a_n) \{ \text{return } E; \}
\]

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) return 1;
    else return x * fact(x-1);
  }
  ```
- Prove: A ⊢ 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
  \( f : \text{int} \to \text{int}, g : \text{int} \to \text{int} \) 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?

\[
\begin{align*}
A \vdash E : T & \quad \text{(return1)} \\
A & \vdash \text{return } E
\end{align*}
\]

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

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