Error Recovery During Type Checking

Supplement to lecture 13

Error Recovery

- As with parsing, it is important to recover from type errors
- Detecting where errors occur is easier than in parsing
  - There is no reason to skip over portions of code
- The Problem:
  - What type is assigned to an expression with no legitimate type?
  - This type will influence the typing of the enclosing expression

Error Recovery Attempt

- Assign type Object to ill-typed expressions

  let y : Int $\leftarrow$ x + 2 in y + 3

  - Since x is undeclared its type is Object
  - But now we have Object + Int
  - This will generate another typing error
  - We then say that that Object + Int = Object
  - Then the initializer's type will not be Int

$\Rightarrow$ a workable solution but with cascading errors

Better Error Recovery

- We can introduce a new type called No_type for use with ill-typed expressions
- Define No_type $\leq C$ for all types C
- Every operation is defined for No_type
  - With a No_type result
- Only one typing error for:

let y : Int $\leftarrow$ x + 2 in y + 3

Notes

- A “real” compiler would use something like No_type
- However, there are some implementation issues
  - The class hierarchy is not a tree anymore
- The Object solution is fine in the class project

One-Pass Type Checking

- COOL type checking can be implemented in a single traversal over the AST

- Type environment is passed down the tree
  - From parent to child

- Types are passed up the tree
  - From child to parent

Implementing Type Systems

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash e_1 + e_2 : \mathrm{Int}} \quad [\mathrm{Add}]$$

TypeCheck(Environment, e_1 + e_2) = {
  T_1 = TypeCheck(Environment, e_1);
  T_2 = TypeCheck(Environment,e_2);
  Check T_1 == T_2 == Int;
  return Int; }