Today
  • Type checking intro

Assignments
  • HW4 out (due Tues)
Type Checking

The goal of type checking is to:

- Detect errors due to type issues, e.g.:

  set x := 0 + "1"       # + requires same-typed operands
  if 42 <= true then    # int can't be compared to bool
    set x := 1
  end

- Also includes “use before def” errors, e.g.:

  var x := 42 + y       # y not defined
  if x > 42 then
    var y := x + 1
  else
    set x := y           # y not defined in this block
  end

- Function call errors, e.g.:

  fun int add(int x, int y)
    return x + y
  end
  var r1 := add(1, 2, 3)  # wrong number of args
  var r2 := add(3.14, 1)  # wrong argument types

Type errors are based on a set of typing rules

- lots of possibilities, different languages have different rules
- we define a set of “strict” typing rules in HW5 (more later)
Basic Approach for HW5

- navigate the AST using the Visitor pattern
- during navigation infer types and look for errors

Given this code ...

1: var int x = 10
2: var r := 0
3: while x > 0 do
4:    set r := r + x
5:    set x := x - 1
6: end

1. For “var int x := 10”
   - check and infer rhs type, compare against declared type, store x’s type
2. For “var r := 0”, infer rhs type, store as r’s type
3. For “while x > 0 do”
   - ensure x is defined and compatible with 0 (both ints)
   - check each body statement ...
4. For “set r := r + x”
   - ensure in rhs that r and x are defined and types are compatible for +
   - ensure lhs is defined and result type (int) is compatible with lhs (r) type
5. etc.

Note we have to keep track of variables and their types!
- we do this using a “symbol table” data structure (var -> type mappings)
Scope and Environment

An environment stores the state of variables within a scope (block)

- We use static (block) scoping in MyPL
- Blocks can be nested (e.g., if-then or while statements)
- The “visibility” of a name is “above” it in file and in an “ancestor” block

“Sub environments” created through while and if statements:

```
# global environment +---------------------+
var x := 1 | x -> int |
while x < 10 do | +-----------------+ |
    # sub environment 1 | | |
    set x := x * 2 | | |
end | +-----------------+ |
if x = 10 then | +-----------------+ |
    # sub environment 2 | | f -> double |
    var f := 3.14 | +-----------------+ |
elif x > 10 then | +-----------------+ |
    # sub environment 3 | | ...
    ...
end +-----------------+
```

To find the type of a given name ...

1. look at the names defined in the current environment first
2. then the parent environment
3. and so on

The “symbol table” maintains the environment information

- which is updated as you navigate the AST
Symbol Table

Stores variable state in a stack of environments as program is being checked

```java
public class SymbolTable {
    public void pushEnvironment();
    public void popEnvironment();
    public bool idExists(string id)
    public void addId(string id);
    public void setInfo(string identifier, Object info);
    public Object getInfo(string identifier);
}
```

- Note that actual implementation will be a bit more complicated ...
- We can add and remove environments (via environment ids) within stack

New environments created/removed when we visit statement lists ...

```java
public void visit(StmtList node) {
    symTable.pushEnvironment();
    for (stmt : node.stmts)
        stmt.accept(this);
    symTable.popEnvironment();
}
```