Static Semantics for MiniJava
Hal Perkins
Winter 2009

Agenda
- MiniJava AST and type checking
- Project overview for semantics phase

Symbol Tables (Recap)
- Build in semantic pass
- Maps names to information
- One per scope, linked to enclosing scope
- Multiple name spaces (classes, methods, variables)
  - So separate map in each symbol table for each namespace

Information About Names
- Different kinds of declarations store different information about their names
  - must store enough information to be able to check later references to the name
- A variable declaration:
  - its type
  - whether it’s final, etc.
  - whether it’s public, etc.
  - (maybe) whether it’s a local variable, an instance variable, a global variable, or ...

Information About Names
- A method declaration:
  - its argument and result types
  - whether it’s static, etc.
  - whether it’s public, etc.
- A class declaration:
  - its class variable declarations
  - its method and constructor declarations
  - its superclass

Generic Type Checking Algorithm
- Recursively type check each of the nodes in the program’s AST, each in the context of the symbol table for its enclosing scope
  - going down, create any nested symbol tables and context needed
  - recursively type check child subtrees
  - on the way back up, check that the children are legal in the context of their parents
Method per AST node class
- Each AST node class defines its own type check method, which fills in the specifics of this recursive algorithm.
  - Generally:
    - declaration AST nodes add bindings to the current symbol table
    - statement AST nodes check their subtrees
    - expression AST nodes check their subtrees and return a result type

Minijava
- Various SymbolTable classes, organized into a hierarchy:
  - SymbolTable
  - GlobalSymbolTable
  - NestedSymbolTable
  - ClassSymbolTable
  - CodeSymbolTable

Symbol Table Operations
- Symbol table classes provide operations such as:
  - declareClass,
  - lookupClass
  - declareInstanceVariable,
  - declareLocalVariable,
  - lookupVariable,
  - declareMethod,
  - lookupMethod

Stored Information
- lookupClass returns a ClassSymbolTable
  - includes all the information about the class's interface
- lookupVariable returns a VarInterface
  - to store the variable's type
  - A hierarchy of implementations:
    - VarInterface
    - LocalVarInterface
    - InstanceVarInterface
- lookupMethod returns a MethodInterface
  - To store the method's argument and result types

Key AST Type Check Operations
- void Program.typecheck() throws TypecheckCompilerExn;
  - type check whole program
- void Stmt.typecheck(CodeSymbolTable) throws TypecheckCompilerExn;
  - type check a statement using a given symbol table
- ResolvedType Expr.typecheck(CodeSymbolTable) throws TypecheckCompilerExn;
  - type check an expression using a given symbol table, returning the type of the result

Forward References
- Type checking class declarations is tricky: need to allow for forward references from the bodies of earlier classes to the declarations of later classes
  ```java
  class First {
    Second next; // must allow this forward ref
    int f() {
      ... next.g() ... // and this forward ref
    }
  }

  class Second {
    First prev;
    int g() {
      ... prev.f() ...
    }
  }
  ```
Supporting Forward References

- So, type check a program’s class declarations in multiple passes
- First pass: remember all class declarations
  \( \text{First} \rightarrow \text{class} \{?, \text{Second} \rightarrow \text{class} \{?\} \} \)
- Second pass: compute interface to each class, checking class types in headers
  \( \text{First} \rightarrow \text{class} \{\text{next:Second}\}, \text{Second} \rightarrow \text{class} \{\text{prev:First}\} \)
- Third pass: check method bodies, given interfaces

Example Type Checking Operation

```java
class VarDeclStmt {
    String name;
    Type type;
    void typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        st.declareLocalVar(type.resolve(st), name);
    }
}
```

- `resolve` checks that a syntactic type expression is legal, and returns the corresponding resolved type
- `declareLocalVar` checks for duplicate variable declaration in this scope

Example Type Checking Operation

```java
class AssignStmt {
    String lhs;
    Expr rhs;
    void typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        VarInterface lhs_iface = st.lookupVar(lhs);
        ResolvedType lhs_type = lhs_iface.getType();
        ResolvedType rhs_type = rhs.typecheck(st);
        rhs_type.checkIsAssignableTo(lhs_type);
    }
}
```

- `lookupVar` checks that the name is declared as a var
- `checkIsAssignableTo` verifies that an expression yielding the rhs type can be assigned to a variable declared to be of lhs type

Example Type Checking Operation

```java
class IfStmt {
    Expr test;
    Stmt then_stmt;
    Stmt else_stmt;
    void typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        ResolvedType test_type = test.typecheck(st);
        test_type.checkIsBoolean();
        then_stmt.typecheck(st);
        else_stmt.typecheck(st);
    }
}
```

- (Garbage collection will reclaim `nested_st` when done)
Example Type Checking Operation

class IntLiteralExpr extends Expr {
    int value;

    ResolvedType typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        return ResolvedType.intType();
    }
}

ResolvedType.intType() returns the resolved int type

Example Type Checking Operation

class VarExpr extends Expr {
    String name;

    ResolvedType typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        VarInterface iface = st.lookupVar(name);
        return iface.getType();
    }
}

Polymorphism and Overloading

- Some operations are defined on multiple types
- Polymorphism occurs when a single operation means and behaves the same while working with different types
  - Ex: Length of a list in ML or such is polymorphic: it doesn’t care what the elements of the list are
  - Ex: Assignment can assign any compatible left-hand and right-hand sides
- Overloading occurs when a single operator has (usually) similar meanings with different implementations
  - Ex: Comparing ints and bools for equality
  - Ex: Ordering ints and strings

Polymorphism and Overloading (cont.)

- Full Java allows methods and constructors to be overloaded, too
- Different methods can have same name but different argument types
- Java 1.5 supports (parametric) polymorphism via generics: parameterized classes and methods
- This all makes type checking more complicated. (So why do we allow it?)

An Example Overloaded Type Check

class EqualExpr extends Expr {
    Expr arg1;
    Expr arg2;

    ResolvedType typecheck(CodeSymbolTable st) throws TypecheckCompilerException {
        ResolvedType arg1_type = arg1.typecheck(st);
        ResolvedType arg2_type = arg2.typecheck(st);
        arg1_type.checkIsInt();
        arg2_type.checkIsInt();
        return ResolvedType.intType();
    }
}

ResolvedType.intType() returns the resolved int type
**MiniJava Project [1]**
- Add resolved type for `double`
- Add symbol table support for static class variable declarations
  - `StaticVarInterface` class
  - `declareStaticVariable` method

**MiniJava Project [2]**
- Add resolved type for arrays: parameterized by element type
- Questions:
  - when are two array types equal?
  - when is one a subtype of another?
  - when is one assignable to another?

**MiniJava Project [3]**
- `ForStmt`
  - loop index variable must be declared to be an `int`
  - test expression must be a `boolean`
- `BreakStmt`
  - must be nested in a loop
- `IfStmt`
  - else statement is optional
- `DoubleLiteralExpr`
  - result is `double`
- `OrExpr`
  - like `AndExpr`

**MiniJava Project [4]**
- `ArrayAssignStmt`
  - array expr must be an array
  - index expr must be an `int`
  - RHS expr must be assignable to array's element type
- `ArrayLookupExpr`
  - array expr must be an array
  - index expr must be an `int`
  - result is array's element type
- `ArrayLengthExpr`
  - array expr must be an array
  - result is an `int`
- `ArrayNewExpr`
  - length expr must be an `int`
  - element type must be a legal type
  - result is array of given element type

**MiniJava Project [5]**
- Extend existing operations on ints to also work on doubles
- Allow unary operations on ints (`NegateExpr`) to be overloaded on doubles
- Allow binary operations on ints (`AddExpr`, `SubExpr`, many others) to be overloaded on doubles
  - Also allow mixed arithmetic: if an int and a double are operands, coerce the int to a double
- Extend `isAssignableTo` to allow ints to be assigned to doubles via implicit coercion

**Where We Are**
- Done with front end of compiler
- Up next: flatten the AST into lower-level intermediate code