An example typechecking operation

class IntLiteralExpr extends Expr {
    int value;

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

ResolvedType.intType() returns the resolved int type

An example typechecking 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

Example: assignment statement: \( \text{lhs} = \text{rhs} \);
- works over any \( \text{lhs} \) & \( \text{rhs} \) types,
  as long as they're compatible
- works the same way for all such types
Assignment is a **polymorphic** operation

Another example: equals expression: \( \text{expr1} == \text{expr2} \);
- works if both exprs are ints or both are booleans
  (but nothing else, in MiniJava)
- compares integer values if both are ints,
  compares boolean values if both are booleans
  (works differently for different argument types)
Equality testing is an **overloaded** operation

Full Java allows methods & 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
An example overloaded typechecking operation

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);
        if (arg1_type.isIntType() && arg2_type.isIntType()) {
            // resolved overloading to int version
            return ResolvedType.booleanType();
        } else if (arg1_type.isBooleanType() && arg2_type.isBooleanType()) {
            // resolved overloading to boolean version
            return ResolvedType.booleanType();
        } else {
            throw new TypecheckCompilerException("bad overload");
        }
    }
}

Typechecking extensions in project (1)

Add resolved type for double

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?

Add symbol table support for static class variable declarations
- StaticVarInterface class
- declareStaticVariable method

Typechecking extensions in project (2)

Implement typechecking for new statements and expressions:
- IfStmt
  - else stmt is optional
- ForStmt
  - loop index variable must be declared to be an int
  - initializer & increment expressions must be ints
  - test expression must be a boolean
- BreakStmt
  - must be nested in a loop
- DoubleLiteralExpr
  - result is double
- OrExpr
  - like AndExpr

Typechecking extensions in project (3)

- 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 int
- ArrayNewExpr
  - length expr must be an int
  - element type must be a legal type
  - result is array of given element type
Typechecking extensions in project (4)

Extend existing operations on ints to also work on doubles

Allow unary operations taking ints (NegateExpr) to be overloaded on doubles

Allow binary operations taking ints (AddExpr, SubExpr, MulExpr, DivExpr, LessThanExpr, LessEqualExpr, GreaterEqualExpr, GreaterThanExpr, EqualExpr, NotEqualExpr) to be overloaded on doubles

• also allow mixed arithmetic: if operator invoked on an int and a double, then implicitly coerce the int to a double and then use the double version

Extend isAssignableTo to allow ints to be assigned/passed/returned to doubles, via an implicit coercion

Type checking terminology

Static vs. dynamic typing

• static: checking done prior to execution (e.g. compile-time)
• dynamic: checking during execution

Strong vs. weak typing

• strong: guarantees no illegal operations performed
• weak: can’t make guarantees

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weak</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Caveats:

• hybrids are common
• mistaken usages are common
• “untyped,” “typeless” could mean “dynamic” or “weak”

Type equivalence

When is one type equal to another?

• implemented in MiniJava with
  ResolvedType.equals(ResolvedType) method

“Obvious” for atomic types like int, boolean, class types

What about type “constructors” like arrays?

```java
int[] a1;
int[] a2;
int[][] a3;
boolean[] a4;
Rectangle[] a5;
Rectangle[] a6;
```

Parameterized types in Java 1.5:

```java
List<int> l1; List<int> l2; List<List<int>> l3;
```

In C:

```c
int* p1; int* p2;
struct {int x;} s1; struct {int x;} s2;
typedef struct {int x;} S; S s3; S s4;
```

Name vs. structural equivalence

Name equivalence:

two types are equal iff they came from the same textual occurrence of a type constructor

• implement with pointer equality of ResolvedType instances
• special case: type synonyms (e.g. typedef) don’t define new types
• e.g. class types, struct types in C, datatypes in ML

Structural equivalence:

two types are equal iff they have same structure

• if atomic types, then obvious
• if type constructors:
  • same constructor
  • recursively, equivalent arguments to constructor
• implement with recursive implementation of equals, or by canonicalization of types when types created then use pointer equality
• e.g. atomic types, array types, record types in ML
Type conversions and coercions

In Java, can **explicitly convert**
   an object of type `double` to one of type `int`
   • can represent as unary operator
   • typecheck, codegen normally

In Java, can **implicitly coerce**
   an object of type `int` to one of type `double`
   • compiler must insert unary conversion operators,
     based on result of type checking

Type casts

In C and Java,
   can explicitly **cast** an object of one type to another
   • sometimes cast means a conversion
     (casts between numeric types)
   • sometimes cast means just a change of static type
     without doing any computation
     (casts between pointer types
      or pointer and numeric types)

In C: safety/correctness of casts not checked
   • allows writing low-level code that’s type-unsafe
   • more often used to work around limitations in
     C’s static type system

In Java: downcasts from superclass to subclass include
   run-time type check to preserve type safety
   • static typechecker allows the cast
   • codegen introduces run-time check
     • Java’s main form of dynamic type checking