1 Part VII: Type Analysis

- Declarations of identifiers is not the only thing to be checked in a compiler.
- Sometimes we have to check for additional properties of the abstract tree.
- Like in many other programming languages the values and variables of Jex have types.
- We have to check that types make sense:
  - The operands of == need to be of same type.
  - The number of arguments in a function call must match the number of formal parameters at the definition.
  - In a variable definition we should really have a type. (It could be another identifier referring to a variable).
  - A function call should really call a function, not a variable.

2 Static or Dynamic Type Checking

- Many of these checks can be done at runtime or at analysis time.
- Jex does only a few checks at analysis time.
- Many other checks are only done at runtime.
  - + has to have two integer arguments and yields an integer result.
  - The condition in an **if** or **while** needs to evaluate to a boolean value.
  - We will have to check all these things in the interpreter.
- Checking at analysis time is called static type-checking.
  - \+ We get error messages already at compile-time.
  - \+ Code runs faster.
- Checking at runtime is called dynamic type-checking.
  - \+ Sometimes more flexible.
  - \+ More intuitive to code?

3 Example: Expressions with booleans

```
Expr = OPERATION Expr Expr op
     | NUMLIT int
     | BOOLLIT boolean
```

Operators are &&, +, ==.

- (3 == 4) && false is considered correct.
- 3 == (4 + true) is considered a type error.

4 Specification of Type Correctness

- First, we have to define, when a program is type correct.
- We cannot do this by a context-free grammar, we have to use other methods.
- Intuitively, every expression has to have a type.
- We formalize this by adding attributes (like type) to the the abstract syntax tree.

```
Expr(t) = OPERATION Expr(t') Expr(t") op
            (op = PLUS && t' = INT && t" = INT && t = INT)
            || (op = AND && t' = BOOL && t" = BOOL && t = BOOL)
            || (op = EQ && t' = t" && t = BOOL)
        | NUMLIT(value) int
            (t = int)
        | BOOLLIT(value) boolean
            (t = boolean)
```

5 Legal Programs

- A program is legal, if
  - it is a sentence in the context-free grammar,
  - there is an attribution for its abstract syntax tree.
- An attribution is an assignment of attributes to the tree, such that all constraints are fulfilled.
- A language is fully described by the context-free syntax and its context dependent syntax.
- Nothing is said about the semantics of the language so far.

6 A typical Language Defintion

Each expression has a *type*.

Operation expression. An operation expression is of the from expression op expression, where op is one of +, &&, and ==.

An operation expression is *type-correct*, if both sub-expressions are type-correct and

- op is + and both sub-expressions have type INT.
- op is && and both sub-expressions have type BOOL.
- op is == and both sub-expressions have equal type.

The *type* of an operation expression is

- INT, if op is +,
- BOOL, if op is && or ==.

7 Implementation in the Visitor

- Here we implement the attribute type as return type in the visitor.
- resultType is a field in the visitor, which is returned by analyze.

```
void caseOperation(Tree.Operation tree) {
    Type t1 = analyze(tree.left);
    Type t2 = analyze(tree.right);
    switch(tree.op) {
        case PLUS:
            if (t1 != INT || t2 != INT) error();
            type = INT;
            break;
        ...
    }
}
```

- The type is propagated from the leaves to the root in the abstract syntax tree. It is therefore called a *synthesized* attribute.
- Synthesized attributes are often implemented as visitor return types.

8 Analysis for Jex

- In Jex we do dynamic type-checking.
- Most checks are done at run-time.
- Only a few checks are done at compile-time:
  - In a variable definition, type should really be a type (not an identifier referring to something else).
  - In **new**, the type should really be a type.
  - In a function definition, the result type should really be a type (The formal parameters are already covered by the variable defintion rule).
  - A function call should really call a function, not a variable or a class
  - A variable reference should never refer to a function.
- Essentially, the different identifiers for functions, variables and classes shouldn't be confused.

9 Context Dependent Syntax of Jex

- We add an attribute sym to FUNDEF, VARDEF, IDENT, and FUNCALL.
- We add an attribute cls to Formal, Type, and IDENT.
- If we found a class in Formal, Type, or IDENT, cls is set to this class.
- Otherwise cls is set to null.


```
Program    = DEFLIST { Definition | Statement }
               "We intepret every part in the global scope.
                We use the global importScope and set toplevel to true"
           ;
Definition = Formal(cls)
           | FUNDEF(sym) Type(cls)  ident { Formal(cls_i) } Statement
               sym instanceof FunSym
               sym.resClass = cls != null
               sym.argClasses[i] = cls_i
               scope.enter(sym)
               "create a new nested scope,
                analyze formals and statement in that scope,
                with toplevel set to false"
           | IMPORT { ident } boolean
               importScope.enter(new Import({ident}, boolean));
           ;
```



```
Formal(cls)  = VARDEF(sym) Type(cls') ident
                   cls = cls' = sym.cls != null
                   (toplevel && sym instanceof GlobalSym
                   || !toplevel && sym instanceof LocalSym)
                   scope.enter(sym)
             ;
Statement    = ASSIGN Expr Expr
             | IF Expr Statement Statement
             | WHILE Expr Statement
             | BLOCK { Statement | Formal }
                   "create a new nested scope, analyze formals
                    and statement in that scope"
             | EVAL Expr
             | RETURN Expr
             ;
```



```
Expr         = NUMLIT int
             | STRINGLIT String
             | BOOLEANLIT boolean
             | IDENT(cls, sym) ident
                   (  (sym = scope.lookup(ident)
                       sym.isVariable()
                       cls = null)
                   || ( scope.lookup(ident) = null
                       cls = importscope.lookup(ident)
                       sym.isClass()
                       cls != null)
                   )
```

```
                 | FUNCALL(sym) ident { Expr }
                          sym.isFunction()
                          sym = scope.lookup(ident)
                 | METHODCALL Expr ident { Expr }
                 | FIELDACCESS Expr ident
                 | OPERATION Expr Expr op
                 | NEW Type(cls) { Expr }
                          cls != null
                 ;
Type(cls)        = IDENT(cls', sym) ident
                            "as for Expr"
                            cls = cls'
                 |  INTEGERTP
                            cls = int.class
                 |  BOOLEANTP
                            cls = boolean.class
                 ;
```

We are informal here in how we treat scope, importscope, and toplevel.

10 We could be more formal

- More formally we could add an attribute scope, importscope, and toplevel to every non-terminal.
- But it is less readable then, because we have many uninteresting rules.
- Imagine the following rule:

```
Expr(sc, is, tl) = OPERATION Expr(sc', is', tl') Expr(sc'', is'', tl'') op
                        sc == sc'
                        sc == sc''
                        is == is'
                        is == is''
                        tl == tl'
                        tl == tl''
```

- But in the implementation we treat them as if we had these attributes.
- We were also sloppy about the order of lookup and enter calls.

11 How do we implement that?

- The function analyze returns now a Java class cls.
- The function analyze in the visitor makes sure, that cls is initialized to null in each visitor.
- If we analyze a type we set cls to the class.
- This can happen in Ident, IntegerTp, or BooleanTp.
- In a variable definition, type should really be a type. So we check whether analyze(tree.tp) != null.
- In **new**, the type should really be a type. So we check whether analyze(tree.tp) != null.
- In a function definition, the result type should really be a type.
- A function call should really call a function. Here we check with sym.isFunction().
- A variable reference should really refer to a variable. Here we check with sym.isVariable().

12 Slots

- There is one more thing, that we would like to do during analysis.
- Each argument or local variable gets a slot assigned, that is some kind of address for the variable at run-time.
- Arguments get negative slot numbers, local variables non-negative slot numbers.
- We implement this by a slot counter nextSlot, which is set to –(number of arguments) at the beginning of a function.
- Then we increment it first for every argument and then for every local variable, that is whenever we analyze a Formal.

13 The Visitor for Name and Type Analysis

```
public class Analyzer implements Tree.Visitor {

    Scope scope;
    ImportScope imports;
    boolean topLevel;

    // next free Slot for local variable
    int nextSlot;

    // class of analyzed Type or Formal
    Class cls;

    // recursive analysis methods
    Class analyze(Tree tree, Scope scope, boolean topLevel) { ... }
    Class analyze(Tree tree, Scope scope) { ... }
    Class analyze(Tree tree) { ... }
```



```
public void caseDefList(Tree.DefList tree) {
    for (int i = 0; i < tree.defs.length; i ++)
        analyze(tree.defs[i]);
}

public void caseFunDef(Tree.FunDef tree) {

    // parameters get negative slot numbers
    nextSlot = - tree.formals.length;

    // the scope starting with the parameters
    Scope paramScope = new Scope(scope);

    // analyse formal arguments
    Class[ ] argClasses = new Class[tree.formals.length];
    for (int i = 0; i < tree.formals.length; i ++)
        argClasses[i] = analyze(tree.formals[i], paramScope);
```



```
    // analyze result type
    Class resClass = analyze(tree.tp);
    if (resClass == null)
        Report.error(tree.pos, "Invalid result type for " + tree.name);

    // symbol for function
    JexSymbol.FunSym fsym = new JexSymbol.FunSym(
        tree.pos, tree.name, tree, resClass, argClasses);
    scope.enter(fsym);

    // analyze body
    analyze(tree.stat, paramScope);

    // nextSlot was incremented for every local variable
    fsym.localCount = nextSlot;
    tree.sym = fsym;
}
```



```
public void caseFunCall(Tree.FunCall tree) {

    // analyze arguments
    for (int i = 0; i < tree.args.length; i ++)
        analyze(tree.args[i]);

    // get function symbol and check
    tree.sym = (JexSymbol) scope.lookup(tree.name);
    if (tree.sym == null) {
        Report.error(tree.pos, "function " + tree.name + " undefined");
    } else if (!tree.sym.isFunction()) {
        Report.error(tree.pos, "calling a non-function " + tree.name);
    }
}
```

```
public void caseIdent(Tree.Ident tree) {
    // get symbol and check
    tree.sym = (JexSymbol) scope.lookup(tree.name);
    if (tree.sym != null) {
        if (tree.sym.isFunction())
            Report.error(tree.pos, "function " + tree.name +
                " used as variable");
    } else {

        // get class and check
        cls = imports.lookup(tree.name);
        if (cls != null)
            tree.sym = new JexSymbol.ClassSym(tree.pos,
                tree.name, cls);
        else
            Report.error(tree.pos, tree.name + " undefined");
    }
}
```


14 From Context-Dependent Syntax to Implementation

- Instead of checking constraints we have to compute the attributes.
- Attributes are usually computed from other attributes.
- Important: Assign attributes only once.

There are different kinds of attributes:

- Some attributes flow up the tree (these are synthesized) e.g. cls.
  - Synthesized attributes are often implemented as return types of the visitor.
- Some flow down the tree (they are inherited) e.g. scope.
  - Inherited attributes are often input parameters to the visitor.

- Some attributes are required to be present later (e.g.sym), they are called persistent.
  - Persistent attributes are stored as additional fields in the abstract syntax tree.
- Others are just required for analysis (e.g. cls) they are called transient.
  - Transient attributes are parameters/result for the visitor.
- Sometimes attributes can be global variables. This may be simpler if arguments change rarely (e.g. importScope).

15 Attribute Grammars

- Context-dependent syntax is sometimes specified unsing an *attribute grammar*.
- This is very similar to the above but completely formal.
- Nodes of the abstract syntax tree are given attributes.
- Attributes are evaluated by assignments, similar to our constraints.
- Attributes are conceptually instance variables in tree nodes.
  - Sometimes attributes are stored in a tree.
  - Sometimes they are stored as global variables.
  - Sometimes they are passed around and not stored.
- There are even tools for them, but in practice they are too complex to use.