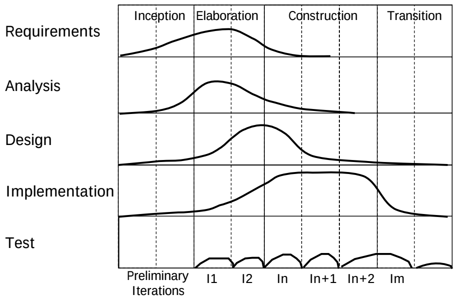
Static Analyzers

Wolfgang Emmerich
Professor of Distributed Computing
University College London
http://sse.cs.ucl.ac.uk

Context



2

Learning Objectives

- To appreciate the basic requirements of static analyzers
- To understand how static analyzers are specified and implemented
- To know the design patterns used in static analyzers
- To understand how static control-flow and data-flow analysis algorithms can be built on top of the object management primitives provided by an IDE

3

Recap: What is static analysis?

- Techniques for checking the static semantics of formal languages
- Implemented in compilers and program editors
- Static analysis techniques are also used to enforce programming conventions and aid design reviews to highlight poor programming practices
- Not constrained to programming languages, e.g.
  - Check correct use of SQL schema in embedded queries
  - Check correct use of XML schemas in web service interface definitions

4

Static analyzers

- Static analysis implemented by static analyzers
- Integral part of any IDE
- Key requirements
  - Efficient calculation of analysis results using
    - Control-flow analysis
    - Data-flow analysis
  - Incremental re-calculation during edits
  - Visualization of static analysis results
  - Derivation of information required for code generation
  - Integration with task management
  - Extensibility (build your own analyzers)

5

Specifying static analyzers

- Parsers are specified by context-free grammars (for example in BNFs or syntax diagrams)
- Context-free grammars are not sufficiently expressive for specifying the context-sensitive and semantic conditions to be checked by a static analyzer
- These are commonly specified using *attribute grammars* that were first introduced by [Knuth 1968]
- Are used extensively in compiler and program editor design

6

Attribute Grammars

- Attribute Grammars (AGs) are an extension of context-free grammars.
- Basic idea:
  - Associate *attributes* with symbols of the grammar
  - Add *semantic rules* to derive attribute values in productions
  - Add *conditions* to productions that check static semantics
- This is very simple and elegant
- Efficient implementations of static analyzers can be derived from AGs automatically for program editors [Reps 1989] or compilers [Kastens et al, 1982]

Attribute Grammar Example

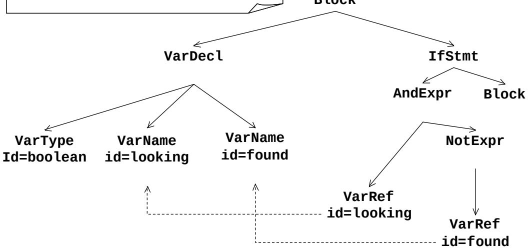
- Consider:

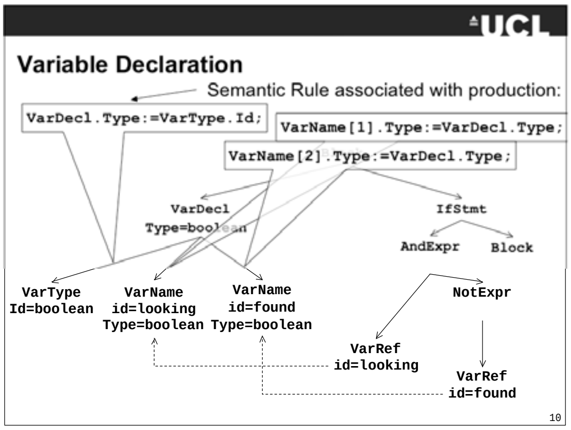
```
boolean looking, found;
…
if (looking && !found ) {…}
```

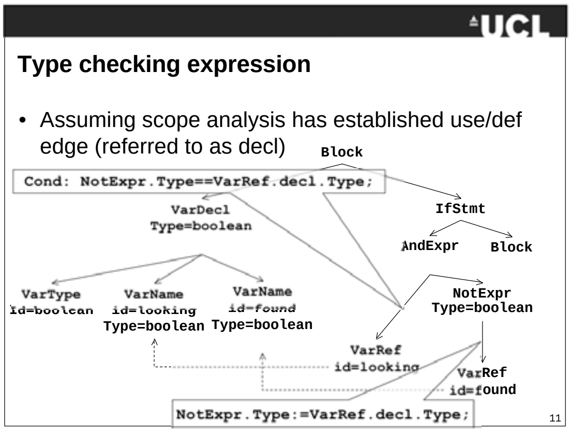
- Aim: check type compatibility within expressions
- Add attribute type to all required symbols
- Semantic rules to derive value of type
- Conditions to check that
  - Type of expression in if statement is boolean
  - Operators used within expression are compatible with arguments

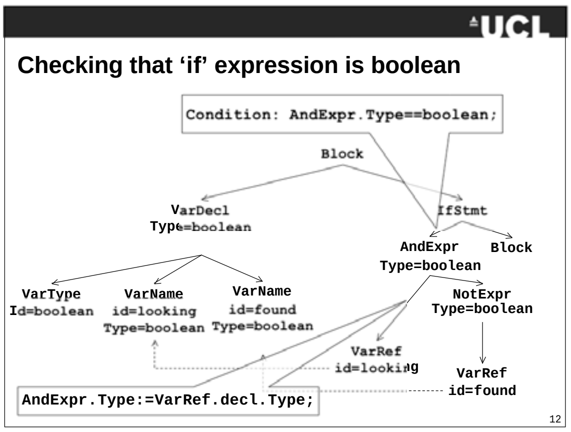
Initial Attributed Abstract Syntax Tree

```
boolean looking, found;
…
if (looking && !found ) {…}
```




UCL
Variable Declaration
Semantic Rule associated with production:
VarDecl.Type:=VarType.Id;
VarName[1].Type:=VarDecl.Type;
VarName[2].Type:=VarDecl.Type;
VarDecl
IfStmt
AndExpr
Block
VarType
Id=boolean
VarName
id=looking
Type=boolean
VarName
id=found
Type=boolean
NotExpr
VarRef
id=looking
VarRef
id=found
10


UCL
Type checking expression
• Assuming scope analysis has established use/def edge (referred to as decl)
Block
Cond: NotExpr.Type==VarRef.decl.Type;
VarDecl
Type=boolean
IfStmt
AndExpr
Block
VarType
VarName
id=looking
Type=boolean
VarName
id=found
Type=boolean
NotExpr
Type=boolean
VarRef
id=looking
VarRef
id=found
NotExpr.Type:=VarRef.decl.Type;
11


UCL
Checking that 'if' expression is boolean
Condition: AndExpr.Type==boolean;
Block
VarDecl
Type=boolean
IfStmt
AndExpr
Type=boolean
Block
VarType
Id=boolean
VarName
id=looking
Type=boolean
VarName
id=found
Type=boolean
NotExpr
Type=boolean
VarRef
id=looking
VarRef
id=found
AndExpr.Type:=VarRef.decl.Type;
12

Well-defined attribute grammars

- An attribute is called:
  - *Inherited* if its value is determined by values of attributes of the parent node (e.g. VarName.Type)
  - *Synthesized* if its value is determined by values of child nodes (e.g. AndExpr.Type)
- The sets of inherited and synthesized attributes are disjoint
- Attribute grammars are well-defined (WAG) iff there is no circular dependency between attributes.
- It is NP complete to decide if an AG is a WAG.

13

Ordered Attribute Grammars

- An AG is ordered AG (OAG) if for each symbol a partial order over the associated attributes can be given, such that in any context of the symbol the attributes are evaluable in an order which includes that partial order.
- Every OAG is a WAG
- It is efficiently decidable whether an AG is an OAG
- An evaluation order for attributes can be calculated automatically in polynomial time.

14

AST Traversals

- AG evaluations need to traverse the AST established by an incremental parser.
- Enrich AST with attributes or semantic links (ASG)
- Execute the semantic rules
- Rule execution needs to be done in the right order
- Requires extensive traversals of the ASG
- Often done using the ‘visitor’ design pattern.

15

Visitor Pattern

- Aim: separate the implementation of an algorithm that traverses a complex structure from the implementation of the structure itself
- Used extensively in IDEs to implement static analysis algorithms that traverse the ASG
- Principle participants:
  - Element Types (i.e. the AST nodes)
  - Visitor (i.e. a base class to permit visits)
  - Concrete Visitor (an implementation of the abstract visitor)

16

Visitor Pattern Example in Eclipse JDT

```
public class  MetricsCalculator {
  int numMethods;
  public int countClasses(){
    ICompilationUnit icu= … // get from the Java Model
    ASTParser parser= ASTParser.newParser(AST.JLS3);
    parser.setSource(icu);
    ASTNode root=parser.createAST(null);
    root.accept(new ASTVisitor() {
      public boolean visit(MethodDeclaration node){
        numMethods++;
        System.out.println("Found class: "+node.getName());
        return true;
      }
    });
  }
}
```

17

Key Points

- Static analyzers specified using attribute grammars
- Evaluation of attribute grammars requires extensive traversals of ASTs
- AST managed by Object Management primitives of the IDE
- Analyzers are separated using visitor pattern

18

References

- D. Knuth. Semantics of context-free languages. Theory of Computer Systems 2(2):127-145. 1968. DOI: 10.1007/BF01692511
- K. Slonneger and B. Kurtz. Formal syntax and semantics of programming languages. Addison Wesley. 1995. www.cs.uiowa.edu/~slonnegr/plf/Book/Chapter3.pdf
- T. Reps and T. Teitelbaum. The Synthesizer Generator Reference Manual. Springer. 1989.
- U. Kastens. Ordered Attributed Grammars. Acta Informatica 13:229-256. 1980.
- U. Kastens et al. GAG: A Practical Compiler Generator. Springer LNCS 141. 1982.
- E. Gamma et al. Design Patterns. Addison Wesley. 1995

19