2.3 Context-Dependent Analysis

Terminology

- Most authors talk about semantic analysis or analysis of the static semantics.
- We use the term context-dependent analysis, because
  - the phase (as the other phases) aims at checking whether the program is a program at all. If it is not a program, it has no semantics.
  - the term semantic analysis applies better to the analysis of the program semantics (e.g. for optimizing, avoiding runtime errors ...)

Learning objectives

- Tasks of context-dependent analysis
- Typical context conditions of programing languages
- Name analysis and relation to binding
- Type analysis and relation to verification
- Techniques for specification and implementation of context-dependent analysis

Task of context-dependent analysis

Check if syntax tree complies to context conditions of language:

- In error case: error handling
- If correct: Provide attributed/decorated syntax tree
Problem of context-dependent analysis

The context-dependent analysis
- creates connection between declaration and applied occurrence of a program element
- checks the correct usage of program elements
- checks further context conditions, e.g. initialization of variables

Examples (2)

Connection to declaration/inheritance:

```java
class A {
    void m() { x = B.x; }
}
static B x;

class B extends A {
}
```

Questions:
- Where is declaration position of x in line 2?
- Where is B in line 2 and 3 declared?
- Where is the declaration of A in line 6?
- Is the static field x inherited, i.e. is the program context correct?

Examples (3)

Usage of declaration and type information: Information computed by the context-dependent analysis is a prerequisite for the translation.

```java
class A {
    void m(Object x) {
        System.out.println( x.toString() );
    }
}
```

Type of xact required to select correct method.

Attribute grammars

- Declare for each non-terminal and each terminal attributes (name and type).
- Declare for each production of the CFG, (semantic) rules how the values of the attributes are computed.

Example: Consider the grammar

```
Prog → AnwSeq,
AnwSeq → e | Anw AnwSeq,
Anw → Ident | AnwSeq
```

Attribute grammar to compute number of occurrences of an identifier in a program:
- non-terminals AnwSeq and Anw get attributes
  - id of type Ident
  - anz of type int
- The semantic rules are as follows:

```
if   =   then 1 else 0
0
128
```
**Attribute grammars (4)**

**Definition**
Let \( \Gamma \) be a CFG. An attribute grammar over \( \Gamma \) consists of:
- two disjoint sets of attributes, \( \text{Inh}(X) \) the set of inherited attributes and \( \text{Syn}(X) \) the set of synthesized attributes.
- an association of attributes to symbols \( X \in T \cup N \) of \( \Gamma \).
- the types/domains of the attributes.
- semantic rules for each inherited attribute \( a \in \text{Inh}(X) \) on the right side of a production and for each synthesized attribute \( a \in \text{Syn}(X) \) on the left side of a production:
  \[
  X_i \cdot a = f(Y_j, \ldots)
  \]
- potentially further semantic actions, e.g. for output.

**Example: Combining productions**

![Diagram of combining productions]

**Attribute grammars (5)**

**Remarks:**
- Synthesized attributes are, for example, the type of an expression or the target code.
- Inherited attributes are, for example, declaration information or symbol tables.
- Different classes of attribute grammars are distinguished by the permitted dependencies among attributes.
- Attribute grammars are generalizations of recursion schemes.
- We use attribute grammars to describe processes.
- Attribute grammars are the specification technique for many compiler generators.

**Example: Attribution of abstract syntax**

- Analogue to attribute grammars
- Attributes are associated to types of abstract syntax
- Attributes of variant types are used for all occurrences of this type, i.e., if \( a \) is an attribute of the variant type \( V \), then each variant \( V_i \) has an attribute \( a_i \).
- Tuple types (and list types) are also attributed.

**Inference rules - concept**

Inference rules are used for specification of static and dynamic language and program properties, in particular for specification of type rules and operational semantics (e.g., structural operational semantics), and is also a basis for language implementation tools.

**Principle:**
- Considered property is formalized with a fixed number of parameters. Each parameter is a program part.
- With inference rules, the property (judgment) is defined inductively over the program structure.

**Example: Language as in the previous examples**

Property: Identifier \( Y \) occurs \( N \) times in program part \( P \) (\( Y \vdash P : N \))

Rules:

\[
\begin{align*}
Y \vdash \text{AS} & : N \\
Y \vdash \text{Prog(AS)} & : N \\
Y \vdash \text{Elem(A, AS)} & : M + N \\
Y \vdash \text{AS} & : N \\
Y \vdash \text{SeqAnw(AS)} & : N \\
Y \not\equiv X & : 0 \\
Y \vdash \text{IdAnw}(X) & : 0 \\
Y \vdash X & : 0 \\
Y \vdash \text{SeqAnw}(X) & : 1
\end{align*}
\]

**Literature**

**Recommended reading:**
Wilhelm, Maurer: Section 9.2, pp. 424-429
2.3.2 Name Analysis

Name analysis is the prerequisite for binding of names to program elements, thus this phase is often called identification.

Program elements are for instance:
- packages, modules, classes, types
- procedures, functions
- variables, fields, records, parameters
- labels

Name analysis determines that each used identifier is also declared.

Checks context conditions
- Has to make declaration information available at applied occurrence of identifier
  - by connecting declaration and applied occurrence
  - by transferring declaration information to applied occurrence

Example: Specification of name analysis

Informal specification technique
- Specify different kinds of named program elements
- Specify defining and applied occurrences of identifiers
- Specify for each declaration which program element is declared and which identifier it binds (binding pair)
- Specify the scope for each programming construct changing scope and which binding pairs are additionally valid and which are hidden
- A program element (PE) with an identifier ID is visible at a program point if the binding pair (ID, PE) is valid and not hidden.

Remark: Specification techniques for name analysis are not as advanced as for context-free analysis (Gap between formal and informal specification techniques).

Example: Specification of name analysis (2)

Program elements: global variables, procedures, formal parameters, local variables, ...
Occurrences of Ident in Var, Proc, LocVar are defining, all other occurrences are applied.
Var defines a global variable, Proc a procedure, LocVar defines either a formal parameter or a local variable, a binding pair consists of a declared identifier and a program element.
2.3.2.2 Realizations based on symboltables

### Name Analysis

#### Attribution for name analysis

- `GlobDeclList, GlobDecl, LocVarList, LocVar, Stat, Exp, ExpList` get inherited attribute `envin` of type `Env`
- `GlobDecl` gets synthesized attribute `envout`

![Diagram showing attribution for name analysis](image)

#### Attribution for name analysis (2)

![Diagram showing attribution for name analysis (2)](image)

#### Attribution for name analysis (3)

![Diagram showing attribution for name analysis (3)](image)

#### Attribution for name analysis (4)

![Diagram showing attribution for name analysis (4)](image)

#### Attribution for name analysis (5)

![Diagram showing attribution for name analysis (5)](image)

#### Attribution for name analysis (6)

- The data type for representing environments and its functional treatment have advantages for specification (easy to understand, more appropriate for verification)
- Specification can be directly implemented, e.g. in functional languages.
- For real compilers, a more efficient technique should be used; problems:
  - Linear search for identifiers is too expensive
  - If hashing is used, values of data type environment cannot be easily represented by sharing, copying of symbol tables would be necessary.

Remarks:

- Example shows basic technique for name analysis.
- Transport of declaration information to applied occurrence is only sketched.
Name analysis with symbol table

Compilation units can contain a large number of identifiers. Thus, insertion and lookup of identifiers should have constant complexity.

Solution:
- Hashing of identifiers in compilation unit (e.g., during scanning); each compilation unit has a finite number of identifiers:
  → use array with constant access
- Enforce sequential access to symbol table such that it can be implemented as a global data structure with destructive updates:
  → avoids copying

Steps for sequential access to symbol table
1. Operations for entering and exiting scope G:
   - Entering of scope G marks declarations of G
   - Exiting G deletes all declarations of G

Steps for development of data structure
1. Since scopes in the considered language (and in most other languages) are nested, entering and exiting scopes can be organized with a stack.
   - For each nesting level nl, store a mapping of identifiers to declaration information.
   - The symbol table is a stack of these mappings.
   - When entering a scope, push an empty mapping onto the stack.
   - When exiting a scope, pop the top-most mapping from the stack.
   - Lookup starts with the top-most mapping. If the identifier is not in the domain, consider next mapping on stack (Complexity: max. nl).

Symbol table for languages with nested scopes

2. Transformation to a finite mapping of identifiers to stacks of declaration information, i.e.
   - stackof (Ident → DeclInf)
   - This can be efficiently implemented by a matrix-linked data structure.

Operations on symbol table
- create(): initializes empty symbol table
- open_scp(s): increments curr_nl
- close_scp(s): deletes all entries for curr_nl, restores identifier links, decrements curr_nl
- enter(id, de, s): adds entry for curr_nl, the entry contains
  - declaration entry (de)
  - pointer for nl links
  - pointer to other entries for the identifier id
- lookup(id, s): access to declaration information of visible binding pair for id

For sequential access in attribution, each operation returns pointer to symbol table. The specified data type is called SymTab.
Example: Usage of SymTab data type

1. Declaration information:
   \[ \text{DeclInf} = \text{VInf} \mid \text{PInf} \]
   \[ \text{VInf}() \]
   \[ \text{PInf}(\text{Int} \text{parcount}) \]

2. Attribution:
   For sequential access to the symbol table, almost all types of the abstract syntax get an inherited attribute \(\text{sym}\) of type SymTab and a synthesized attribute \(\text{symout}\). Additionally, \(\text{LocVarList}\) and \(\text{ExpList}\) get an synthesized attribute \text{length} of type \text{Int}.

Example: Semantic rules (2)

Example: Semantic rules (4)

Example: Semantic rules (5)

Declaration information

Declaration information has to be sufficient
- to check the correct application of an identifier
  - Is a variable allowed at the applied occurrence?
  - Is there a procedure, type, label ... allowed?
  - Is the number of parameters correct?
  - Are the types correct?
- to handle named scopes
  - Is a selector admissible?
  - Which methods and fields are contained in a class?
- to check additional context conditions (e.g. case statement in Pascal)
### Declaration information (3)

**Remarks:**
- Name analysis and its implementation depend on the rules of the programming language, e.g. name spaces, application of declarations, overloading.
- For separate translation, a symbol table for each translation unit should be created that contains the declaration information of the program elements used by other translation units.
- The symbol table is also used for distributing other information in later phases, e.g. address information of variables.

### Literature

**Recommended Reading:**
- Wilhelm, Maurer: Sections 9.1.1 and 9.1.2, pp. 408 – 416
- Appel: Section 5.1, pp. 108 – 119

### Implementation of attribution

**Learning objectives:**
- Different interpretations of attributions
- Introduction into the semantics of attributions
- Classes of attribute grammars
- Implementation techniques

### Interpreting of attribute grammars

1. **Specification and equation interpretation**
   An attribute grammar (AG) assigns attribute values to nodes of grammar trees. The specification is given by an equation system that is generated for each tree.
   
   **Question:** Has the system a (unique) solution?

2. **Functional or computational interpretation**
   An AG determines how attribute values in a tree are computed.
   - w/o storage: goal is computation of attribute values at root node only
   - with storage: goal is computation of all attribute values for further processing phases

### Classes of attribute grammars

**Definition (Non-circular attribute grammar)**
Let $AG$ be an attribute grammar over $CFG \, \Gamma$. For syntax trees $S$ of $\Gamma$, let $D(S)$ denote the directed attribute dependency graph. The $AG$ is non-circular if $D(S)$ is cycle free for all $S$.

**Remarks:**
- For non-circular attribute grammars, the equation system for each syntax tree has a unique solution.
- Testing non-circularity is exponential in general. Thus and for generating efficient attribute evaluators, restricted classes of attribute grammars are considered.

**Classes of attribute grammars:**

- $S$-attributed if $AG$ only has synthesized attributes
- $L$-attributed if the attributes for each syntax tree for $\Gamma$ can be evaluated with a left-right depth-first traversal.

Other important classes of attribute grammars:
- absolute non-circular (Kennedy & Warren)
- ordered (Karstens)
Implementation of attribute grammars

1. Computational interpretation w/o storage

For L-attribute grammars and other simple classes of attribute grammars, attribute evaluation can be performed by recursive functions:

- Inherited attributes become parameters
- Synthesized attributes become results

For each production, a function is defined that calls the semantic actions and the functions for the children (sometimes one function can handle several productions).

Remark: For other attribute grammar classes, several functions per production might be used.

Example: L-AG implemented by recursive functions

Consider simple L-AG (cf. lecture): The environment parameter represents the inherited attribute, the result represents the synthesized attribute:

```
emptyST = SymTab()
lookup(_,_) = UNDEF ?
lookup(_,_) = METH ?
lookup(_,_) = FIELD ?
```

Signature of symbol table

- `emptyST : SymTab`
- `lookup : Ident x SymTab -> FieldMethUndef`
- `enter : Ident x FieldMeth x SymTab`

Example: Attribute evaluator (4)

Interface of symbol table

```java
class SymTab {
    static int FIELD = 1;
    static int METH  = 2;
    static int UNDEF = 3;
    static SymTab emptyST() { return new SymTab(); }

    int lookup(String id) { // liefert SymTab.FIELD oder ...
        SymTab.enter(String id, Boolean isField) {
            // ...
        }
    }
}
```

Example: Attribute evaluator (5)

Attribute declaration and 2-pass evaluation

```java
class XClass {
    DeclList dl;
    // keine Attribute
    void evalAttr() {
        dl.stIn = SymTab.emptyST();
        dl.evalAttr();
        dl.stOut;
        dl.evalAttr();
    }
}
```
Example: Attribute evaluator (6)

```java
abstract class FieldDecl extends Decl {
    String id;
    void evalAttr0() {
        if( this.stIn.lookup(id) != SymTab.UNDEF )
            so.println(“Identifier already declared!”);
        this.stOut = this.stIn.enter(dmid,false);
        dc.evalAttr1();
    }
}
```

Example: Attribute evaluator (7)

```java
abstract class Decl extends Decl {
    String uaid;
    void evalAttr1() {
        if( this.st.lookup(umid) != SymTab.METH ){
            so.println(“Method not declared!”);
        } else{
            this.stOut = this.stIn.enter(dmid,false);
        }
    }
}
```

Example: Attribute evaluator (8)

```java
class FieldDecl extends Decl {
    String id;
    void evalAttr0() {
        if( this.stIn.lookup(id) != SymTab.UNDEF )
            so.println(“Identifier already declared!”);
        this.stOut = this.stIn.enter(dmid,false);
    }
}
```

Remark and literature

**Remark:** Attribution and attribute evaluation are techniques to decorate tree structures and are not limited to context-dependent analyses of languages.

**Recommended Reading**

Wilhelm, Maurer: Sections 9.2 and 9.3, pp. 424–436

Type analysis

**Relevance of Typing**

- improves readability of programs
- improves possibilities for static checks
- allows more efficient implementations, because runtime checks can be avoided and memory can be allocated more specifically
- prerequisite for high-level programming techniques

Tasks of Type Analysis

- Computation and inference of explicit type information
  - for declared program elements often from declaration information
  - for expressions from sub-expressions
  - potentially complex algorithm (because of overloading, generic types, missing type declarations)
- Type correctness: check of typing rules

2.3.3 Type Analysis

2.3.3.1 Specification of type analysis
Specification of type analysis

The type system of a programming language determines
- which types/type schemes exist in the language
- how types and type instances relate
- which rules a typed program has to satisfy to be type correct

For a formal specification of a type system, types are described by an
abstract syntax and type rules by inference rules.

Example: Type System

Java (w/o generics) has the following types:

```
| JavaType = PrimitiveType | ReferenceType
PrimitiveType = byte | short | int | long |
ReferenceType = ClassType(Ident) | InterfaceType(Ident) |
| ArrayType(JavaType) |
```

The primitive types are pre-defined. Class and interface types are user
defined. Field types are defined implicitly.

Note: A program has infinitely many types.

Type rules

Consider the following subset of Java:

```
Stat  = IfStat | Invoc | Assign
IfStat = Exp cond, Exp then, Exp else
Invoc = Exp target, Ident name, Exp par
Assign = Ident lhs, Exp rhs
Exp = Assign | Invoc | Arithm | Relation
Arithm = Exp le, ArithmOp op, Exp re
Relation = Exp le, RelOp exp, Exp re
Cast = JavaType t, Exp e
Const = IntConst | LongConst
Var = Ident id
Null {} |
```

Definition of subtyping relation in Java:

S is a subtype of T, S ≤ T, iff

- S = T or
- S and T are reference types and
  - S extends or implements the type of T or
  - T = Object or
- S = ArrayType(SE) and T = ArrayType(TE) and SE ≤ TE or
- there exists type U with S ≤ U and U ≤ T
Type Analysis

Type Analysis

Syntax and Type Analysis

90

Relation(e1, rop, e2) : boolean

e1 : T, e2 : T

Arithm(e1, aop, e2) : T

e1 : T, e2 : ... b

ClassType(MeinTyp) ],
ClassType(MeinTyp)  & InterfaceType(EinTyp),
...               }

Bemerkungen:
• Typsicherheit kann auch durch Laufzeitprüfungen
und entsprechende Fehlerbehandlung erreicht
und entsprechende Fehlerbehandlung erreicht
... A. Poetzsch-Heffter, TU Kaiserslautern
10.05.2007

// ArrayStoreException
strfeld[0].length();

Bemerkungen:
• Typsicherheit kann auch durch Laufzeitprüfungen
und entsprechende Fehlerbehandlung erreicht
und entsprechende Fehlerbehandlung erreicht
... A. Poetzsch-Heffter, TU Kaiserslautern
10.05.2007

// ArrayStoreException
strfeld[0].length();

Context-Dependent Analysis

Context-Dependent Analysis

Typed Languages - Example

Terminology

• A program is type correct if it satisfies the type rules.
• A type error occurs if
  • a value of an incorrect type is assigned to a variable
  • an operation is called with an inadmissible parameter
• A program is type safe if its execution does not lead to type errors.
• A programming language is strongly typed if type correctness implies type safety. (This is a proof of a semantic program property. ⇒ semantic analysis)

Typed Languages - Example

Task:

Zeige: \( \Sigma \models Assign( a, Assign( b, Null() ) ) : InterfaceType(EinTyp) \) mit
\( \Sigma = \{ [ a \rightarrow InterfaceType(EinTyp) ],
[ b \rightarrow ClassType(MeinTyp) ],
ClassType(MeinTyp) \leq InterfaceType(EinTyp),
... \} \)

Example: Featherweight Java

Specification of name and type analysis in Featherweight Java

A. Igarashi, B.C. Pierce, P. Wadler: Featherweight Java: A Minimal Core Calculus for Java and GJ, ACM Transactions on Programming Languages and Systems, 23(3), May 2001

2.3.3.2 Type safety

Type Safety

A type characterizes specific properties of program elements and constructs, e.g.
• a variable of type T is only for storing values of type T
• an expression of type T only provides values of type T
Types can also carry semantic information.

Example: Guaranteed Program Properties

• A variable of type A contains only values of type A.
• Evaluation of an expression of type A only yields values of type A.
• If a method m is called, its existence in the target object is ensured.

Remarks:
• Type safety can also be obtained by runtime checks and error handling.
• Type safety ensures the safe application of operations and is the prerequisite of further safety checks.

2.3.3 Implementation aspects
Implementation Aspects

Techniques and algorithms for type analysis strongly depend on the concrete language:
- For many languages, type inference can be implemented by S-attribution.
- For languages with polymorphism, type analysis can be arbitrarily complex or even undecidable.

Procedure:
1. For each program construct to be typed, determine exactly one type (type inference)
2. Check context conditions (type checking)
   - Type of conditions in a conditional statement or loop has to be boolean.
   - Type of current parameter expressions has to be compatible with formal parameter types.
   - Implicit type conversions have to be possible.

Type Analysis

Consider previously defined Java subset.

1. Data Types for Attribution
   For representing type information:
   \[ JType = \text{JavaType} | \text{NullType} \]
   where NullType() is used to type the constant null and is a subtype of all reference types.

where
\[
\text{max}(S, T) = \begin{cases} 
T & \text{if } S \leq T \\
S & \text{if } T \leq S \\
\text{undefined} & \text{else}
\end{cases}
\]
and \( \text{isCompatible}(S, T) \iff S \leq T \) or \( S \) is convertible to \( T \).

Type Analysis - Example

For representing environment information, we use symbol tables of type SymTab with the following operations:
- lookupVar: Ident \times SymTab \rightarrow JType
- lookupMth: Ident \times Ident \times SymTab \rightarrow (Signature | undefined)
- isCompatible: JType \times JType \rightarrow boolean
- max: JType \times JType \times SymTab \rightarrow (JType | undefined)

2. Attribution:
   Statements and expressions get the inherited attribute SymTab.
   Expressions get the synthesized attribute JType which are denoted by:
   \[ \text{SymTab} \text{ stab} \]
   \[ \text{JType} \text{ typ} \]

Note: For the attribution, we assume that name analysis has already been completed.
2.3.4 Interplay of name and type analysis

Name and type analysis are in general mutually dependent. For instance:
- For expressions of the form e.a, the type of e is required or name analysis of a (analog for method calls).
- For overload resolution of procedure or method names, the types of the current parameters are required.

In particular, rules and algorithms for overload resolution are strongly language dependent.

Overload Resolution in Java

Goals:
- Example of language report, i.e. basis of language implementation
- Connections between context-dependent analysis
- Learn overloading rules of a commonly used language

12.12.2.2 Choose the Most Specific Method

If more than one method declaration is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the most specific method is chosen.

The informal intuition is that one method declaration is applicable to a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is more specific.

For instance:
- A method is said to be maximally specific for a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is more specific.
- If there is exactly one maximally specific method, then it is in fact the most specific method.

Recommended Reading:
- Appel, Sections 5.2, 5.3, pp. 116 – 127

Connections between Context-Dependent Analyses

Name and type analysis are in general mutually dependent. The attribution is deterministic in contrast to the specification.
Overload Resolution (Java (3))

```java
class Point {
    int x, y;
}
class ColoredPoint extends Point {
    int color;
}
class Test {
    static void test(ColoredPoint p, Point q) {
        out.println("(ColoredPoint, Point)\n");
    }
    static void test(Point p, ColoredPoint q) {
        out.println("(Point, ColoredPoint)\n");
    }
    static void test(ColoredPoint p, ColoredPoint q) {
        out.println("(ColoredPoint, ColoredPoint)\n");
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        test(cp, cp); // Aufrufstelle
    }
}
```

The third declaration is most specific at the marked applied occurrence. If it was missing, the call would be erroneous and cause a translation error.

---

*Recommended Reading:*

- Wilhelm, Maurer: Section 9.1.3, pp. 416 – 419