Honors Compilers

Semantic Analysis and Attribute Grammars

Mar 5th 2002
Semantic analysis

Enforce context-dependent language rules that are not reflected in the BNF, e.g. a function must have a return statement.

Decorate AST with semantic information for subsequent code generation, e.g. determine types of all expressions.

Expand complex constructs in preparation for code generation, e.g. aggregates into loops.

General framework: compute attributes
Attributes and attribute grammars

Syntax-directed framework:

- For every symbol in the grammar we define some computable properties (e.g. the value of a constant expression)
- For every production in the grammar we give computation rules for the properties of all symbols on both sides of the production (e.g. the value of a sum is the sum of the values of the operands)
- The rule is local: it only refers to other symbols in the same production
- The evaluation of the attributes can require an arbitrary number of traversals of the AST: arbitrary context dependence (e.g. the value of a declared constant is found in the constant declaration)
Example: binary strings

String ::= Digit | String Digit

Digit ::= 0 | 1

Attribute: the numeric value of a string

Digit ::= ‘0’

Val \text{digit} := 0;

Digit ::= ‘1’

Val \text{digit} := 1;

String ::= Digit

Val \text{String} ::= Val \text{digit}

String ::= String Digit

Val \text{String1} := 2 * Val \text{String2} + Val \text{digit}
Example: type imposed by context

Assignment ::= Name := Expression

Attribute: if expression is overloaded, its type is determined by the type of the name:

context_Type expression := Type name

Two distinct attributes: type of name is determined from context and/or other rules
Inherited and synthesized attributes

If attribute of left-hand side is computed from attributes in the right-hand side, attribute is **synthesized**: bottom-up propagation

If attribute of symbol on right-hand is computed from attributes of left-hand side, or from attributes of other symbols on right-hand side, attribute is **inherited**: top-down propagation of information
General results

Attribute grammars have the power of Turing machines.

Attributes are computed by repeated passes over the AST.

Attribute definitions may be cyclic; checking whether an attribute grammar has cycles is decidable but potentially expensive.

In practice inherited attributes are handled by means of global data structures (symbol table).

Useful subsets: L-attributed and S-attributed grammars.
In an L-attributed grammar, inherited attributes can be computed left-to-right:

\[ N ::= S_1 \ S_2 \ S_3 \]

attributes of \( S_1 \) cannot depend on attributes of \( S_2 \) or \( S_3 \)

Easy to implement in a top-down parser: when building a node, nodes on which its attributes depend have been seen and processed.

An S-attributed grammar has only synthesized attributes

Usable with bottom-up parser

Can convert L- to S- but result is awkward.
Some important attributes

For expressions: **type**
For overloaded calls: **candidate interpretations**
For identifiers: **entity** (defining_occurrence)
For definitions: **scope**
For data/function members: **visibility** (public, protected, private)
For function: **virtual functions** (primitive operations)
Etc, etc.
Attribute computation and tree traversals

In gnat (and others) mostly left-to-right, top-down traversal, with localized multiple traversals

Inherited attributes computed during declaration processing, symbol table carries inherited information as one global data structure

Synthesized attributes on terminals: names, literal values

In the presence of overloading, type is both inherited and synthesized: two passes required over expressions.

Generated code can be treated as synthesized attribute
Name Resolution

Compute attribute entity: associate every identifier (use occurrence) with the corresponding defining occurrence.

If entity is overloaded, associate entity with set of candidate entities, to be resolved by types and context.

Complications:

- Block structure and hiding rules
- Context and import rules
Name resolution and block structure

Basic rule: **inner definition hides outer one with same name**

Data structures reflect scope nesting

- A stack of scopes: defining occurrences of functions, packages, blocks, loops, records
- A list of local entities declared in each scope
- A names table

Entry in names table points to innermost occurrence of entity with given name

All identifiers with given name point to same names table entry *(handled by scanner)*

Name resolution does not require any hashing
Data structures for name resolution

Entity chain, homonym chain, chars
A **scope** is any entity that has local declarations: package, function, record type, task, block.

**On scope entry:** place new scope on stack, initialize list of local entities

**For every declaration:** chain name entry to local entity, set homonym of local entity to outer entity with same name

**On scope exit:** chain name entry to homonym of local entity. Local entity becomes invisible.

**On package exit:** leave entries chained, because entities remain visible through qualified notation.

Full information remains in the tree for subsequent passes.
Resolving qualified names

To resolve A.B, first resolve A (direct name). If A is enclosing scope, follow **homonym chain** for B until we find a variable whose scope is A.

If A is a variable, find its type:
- If **record** or **struct**, find component of type named B
- If **pointer**, apply rule to designated type (implicit dereference)
- If **task**, find entry named B

To resolve A.B.C, **recurse**: resolve prefix A.B, then apply previous rules.

To resolve A->B (C++): type of A must be of the form *T, proceed as above.
Packages (namespaces) are scopes, have chain of local entities, but are not in scope stack.

Package entities also linked to names table

To resolve P.A, if P is a package in context (with_clause) use same algorithm as for an enclosing scope

If use_clause is active for P, then A is a legal use of P.A  Treat as a direct name if there is no A in an enclosing scope that hides it: entities in enclosing scopes hide imported entities.

If multiple entities are use-visible, reference is ambiguous
Detailed algorithms in gnat sources

Visibility : sem_ch8

- **New_Scope**: push current scope on scope_stack
- **Find_Direct_Name**: resolve identifier
- **Find_Selected_Component**: resolve A.B, where A can be an expression
- **Find_Qualified_Name**: resolve A.B, where A is a scope
- **Use_One_Package**: make entities that are local to a package use-visible in current context
- **End_Use**: when use clause goes out of scope, the converse
- **End_Scope**: remove local entities from visibility, pop scope_stack

Declarations: sem_ch3, sem_util

- **Enter_Name**: enter new entity into local chain, diagnose duplicates
Top-down processing: all but expressions

Semantic analysis of package declaration:
- Enter new scope
- Process visible declarations
- Process private declarations
- Exit scope

Semantic analysis of while statement
- Process (analyze and resolve) condition
- Process list of statements

Semantic analysis of object declaration
- Enter new entity into current scope
- Resolve type definition
- Analyze and resolve expression