CSE 401 – Compilers

Lecture 13: Semantic Analysis, Part I

Michael Ringenburg

Winter 2013

Reminders

- Project part 2 due in 1 week (Wednesday, February 13).
- Midterm a week from this coming Friday (Friday, February 15).
  - Midterm will cover material up to and including this Friday's lecture (February 8). So, scanning, parsing, and some semantic analysis. I.e., the compiler front end.

Agenda For The Next Week

- Static semantic analysis
  - What properties can we check/enforce, and how?
  - What else can we glean about the program from walking the AST?
- Types
- Constant Folding
- Attribute grammars
- Representing types
- Symbol tables
- You will need some of this for project part 3

Example: What do we need to check to compile this?

```
class C {
    int a;
    C(int initial) {
     a = initial;
    }
    void setA(int val) {
     a = val;
    }
}
```

```
class Main {
  public static void
    main(String[] args) {
      C c = new C(17);
      c.setA(42);
    }
}
```

Beyond Syntax

- There is a level of correctness that is not captured by a context-free grammar
  - Has a variable been declared?
  - Are types consistent in an expression?
  - In the assignment x=y, is y assignable to x?
  - Does a method call have the right number and types of parameters?
  - In a selector p.q, is q a method or field of class instance p?
  - Is variable x guaranteed to be initialized before it is used?
  - In p.q, could p be null?
  - Etc.

Checked Properties

- Some enforced at compile time, others at run time (typically depends on language spec).
- Different languages have different requirements
  - E.g., C vs. Java typing rules, initialization requirements
  - Some of these properties are often desirable in programs, even if the languages doesn't require them.
  - Compilers shouldn't enforce a property that is not required by the language (but can warn).
  - However, there are static checkers for some of these properties that use compiler-style algorithms.

What else do we need to know to generate code?

- Where are fields allocated in an object?
- How big are objects? (i.e., how much storage needs to be allocated by new)
- Where are local variables stored when a method is called?
  - Stack? What offset? Or exclusively in register? Which?
  - Aside: what happens to registers when a method is called?
- Which methods are associated with an object/class?
  - In particular, how do we figure out which method to call based on the run-time type of an object?

Semantic Analysis

- Main tasks:
  - Extract types and other information from the program
  - Check language rules that go beyond the context-free grammar
  - Resolve names
    - Relate declarations and uses of each variable
  - "Understand" the program well enough for synthesis
    - E.g., sizes, layouts of classes/structs
- Key data structure: Symbol tables
  - Map each identifier in the program to information about it (kind, type, etc.)
- This is typically considered the final part of the "front end" of the compiler (once complete, know whether or not program is legal).

Constant Folding

- *Constant folding* is a simple optimization that computes at compile-time the results of operations whose operands are all constants (e.g., integer literals)
- It is often applied many times during compilation, as certain other optimizations may reveal additional folding opportunities
  - E.g., *constant propagation* may cause a variable access to be replaced with a constant.
- Many compilers perform the first pass during the front end semantic analysis phase.
  - Can be done via a depth-first traversal of the AST.

Example: AST and depth-first folding traversal

```
x = 1 + 2;
…
y = (2*5 + 5)/x;
```

During later optimization/analysis phases, we may be able to prove that x does not change between the two statements, in which case we could *propagate* the folded constant value of x forward to it's usage in the computation of y's value. After that propagation, a subsequent folding phase could then eliminate the division.

Some Kinds of Semantic Information

Information	*Generated From*	*Used to process*
Symbol names (variables, methods)	Declarations	Expressions, statements
Type information	Declarations, expressions	Operations
Memory layout information	Assigned by compiler	Target code generation
Values	Constants	Expressions

What do we need for semantic checking?

- For each language construct we want to know:
  - What semantic rules should be checked
    - Specified by language definition (type compatibility, required initialization, etc.)
  - For an expression, what is its type (used to check whether the expression is legal in the current context)
    - Computing the type of an expression is sometimes referred to as "inferring the type" (although this an overloaded term).
  - For declarations, what information needs to be captured to use elsewhere

A Sampling of Semantic Checks and Computations

- Appearance of a name in an expression: id
  - Check: Symbol has been declared and is in scope
  - Compute: Inferred type is the declared type of symbol
- Constant: v
  - Compute: Inferred type and value are explicit
  - Example: **`42.0`** has type double and value 42.0

A Sampling of Semantic Checks and Computations

- Binary operator: exp_1 op exp_2
  - Check: exp_1 and exp_2 have compatible types
    - Either identical, or well-defined conversion to appropriate types
    - Types are compatible with op
    - Example: **`42 + true`** fails, **`20 + 21.9999`** passes
  - Compute: Inferred type of expression is a function of the operator and operand types
    - Example: **`20 + 21.999`** has type double, **`42 + ", the answer"`** has type String (in Java).

A Sampling of Semantic Checks and Computations

- Assignment: exp_1 = exp_2
  - Check: exp_1 is assignable (not a constant or expression)
  - Check: exp_1 and exp_2 have (assignment-)compatible types
    - Identical, or
    - Type of exp_2 can be (automatically) converted to exp_1 (e.g., char to int), or
    - Type of exp_2 is a subclass of type of exp_1 (can be decided at compile time)
    - Example: **`x + 5 = 4`** fails, **`x = 42`** passes if x in an integer or double, fails if x is a boolean
    - Ex: **`Object a = new Integer(); Number b = a;`** also fails (a's *static* type not a subclass of b's type).
  - Compute: Inferred type is type of exp_1

A Sampling of Semantic Checks and Computations

- Cast: (exp1) exp2
  - Check: exp1 is a type
  - Check: exp2 either
    - Has same type as exp1
    - Can be converted to type exp1 (e.g., double to int)
    - Downcast: is a superclass of exp1
      - May generate a runtime error is exp2 isn't really an exp1, e.g., **`animal a = new animal(); dog d = (dog)a;`** where dog extends animal.
    - Upcast: is the same or a subclass of exp1
  - Compute: Inferred type is exp1

A Sampling of Semantic Checks and Computations

- Field reference: exp.f
  - Check: exp has a reference type (class instance)
  - Check: The class of exp has a field named f
  - Compute: Inferred type is declared type of f

A Sampling of Semantic Checks and Computations

- Method call: exp.m(e_1, e_2, …, e_n)
  - Check: exp is a reference type (class instance)
  - Check: The class of exp has a method named m
  - Check: The method exp.m has *n* parameters
  - Check: Each argument has a type that can be assigned to the associated parameter
    - "Assignment compatible", like our assignment checking
  - Compute: Inferred type is given by method declaration return type (possibly void)

A Sampling of Semantic Checks and Computations

- Return statement: "return exp;" or "return;"
- Check:
  - If the method is non-void:
    - The expression can be assigned to a variable with the declared return type of the method (if the method is not void) – exactly the same test as for assignment statement
  - If the method is void:
    - There's no expression
- Don't infer types of statements (just expressions)

Attribute Grammars

- A systematic way to think about semantic analysis
- Formalize properties checked/computed during semantic analysis and relate them to grammar productions in the CFG.
- Sometimes used directly, but even when not, AGs are a useful way to organize the analysis and think about it

Attribute Grammars

- Idea: associate attributes with each node in the syntax tree
- Examples of attributes
  - Type information
  - Storage information
  - Assignable (e.g., expression vs variable – lvalue vs rvalue for C/C++ programmers)
  - Value (for constant expressions)
  - etc. …
- Notation: X.a if a is an attribute of node X

Attribute Example: (1+2) * (6 / 2) * x

Given exp ::= INT
Let exp.val = INT
Given exp ::= id
Let exp.val = UNK
Given exp ::= exp_1 **<op>** exp_2
Let exp.val = exp_1.val **<op>** exp2.val
Where UNK **<op>** INT =
INT **<op>** UNK =
UNK

Next time

- More on attribute grammars, plus a deeper example.
- Symbol Tables (and symbol tables for MiniJava compilers).