CSE 401 – Compilers

Static Semantics

Hal Perkins

Winter 2009

Agenda

- Static semantics
- Types
- Symbol tables
- General ideas for now; details later for MiniJava project

What do we need to know to compile this?

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

```
class Main {
  public static void main(){
        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?
  - Could p be null when p.q is executed?
  - Etc. etc. etc.

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?
- 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 assignments to and references of each variable
  - “Understand” the program well enough for synthesis
- Final part of the analysis phase / front end of the compiler

Symbol Tables

- Key data structure during semantic analysis
  - For each identifier in the program, record its attributes (kind, type, etc.)
  - Later: assign storage locations (stack frame or object offsets) for variables; other annotations
- Build during semantics pass
  - Maps identifier names to information
  - Declarations add bindings to table
  - Uses look up information – error if not found

Nested Scopes

- Can have same name declared in different scopes
  - Why?
- References use closest textually-enclosing declaration
  - static/lexical scoping, block structure
  - closer declaration shadows declaration of enclosing scope

Nested Scopes: Approach

- Simple solution
  - one symbol table per scope
  - each scope's symbol table refers to its lexically enclosing scope's symbol table
  - root is the global scope's symbol table
  - look up declaration of name starting with nearest symbol table, proceed to enclosing symbol tables if not found locally
- All scopes in program form a tree
- Industrial-strength compiler: engineer this so table operations are O(1)

Name Spaces

- One name may unambiguously refer to different things

```
class F {
  int F(F F) {// 3 different F's
     ... new F() ...
     ... F = ...
     ... this.F(...) ...
  }
}
```

- MiniJava has three name spaces: classes, methods, and variables
  - We always know which we mean for each name reference, based on its syntactic position
  - So, have the symbol table store a separate map for each name space

Some Kinds of Semantic Information

Information	*Generated From*	*Used to process*
Symbol tables	Declarations	Expressions, statements
Type information	Declarations, expressions	Operations
Constant/variable information	Declarations, expressions	Statements, expressions
Register & memory locations	Assigned by compiler	Code generation
Values	Constants	Expressions

Semantic Checks

- For each language construct we want to know:
  - What semantic rules should be checked: specified by language definition (type compatibility, etc.)
  - For an expression, what is its type (used to check whether the expression is legal in the current context)
  - For declarations in particular, what information needs to be captured to be used elsewhere
- Following slides: A sampler
  - Not specific to the project (we'll do that later)

A Sampling of Semantic Checks (0)

- Name use: id
  - id has been declared and is in scope
  - Inferred type of id is its declared type
  - Memory location assigned by compiler
- Constant: v
  - Inferred type and value are explicit

A Sampling of Semantic Checks (1)

- Binary operator: exp_1 op exp_2
  - exp_1 and exp_2 have compatible types
    - Identical, or
    - Well-defined conversion to appropriate types
  - Inferred type is a function of the operator and operands

A Sampling of Semantic Checks (2)

- Assignment: $exp_1 = exp_2$
  - exp_1 is assignable (not a constant or expression)
  - exp_1 and exp_2 have compatible types
    - Identical, or
    - exp_2 can be 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)
  - Inferred type is type of exp_1
  - Location where value is stored is assigned by the compiler

A Sampling of Semantic Checks (3)

- Cast: (exp_1) exp_2
  - exp_1 is a type
  - exp_2 either
    - Has same type as exp_1
    - Can be converted to type exp_1 (e.g., double to int)
    - Is a superclass of exp_1 (in general requires a runtime check to verify that exp_2 has type exp_1)
  - Inferred type is exp_1

A Sampling of Semantic Checks (4)

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

A Sampling of Semantic Checks (5)

- Method call exp.m(e_1, e_2, …, e_n)
  - exp is a reference type (class instance)
  - The class of exp has a method named m
  - The method has *n* parameters
  - Each argument has a type that can be assigned to the associated parameter
  - Inferred type is given by method declaration (or is void)

A Sampling of Semantic Checks (6)

- Return statement return exp; return;
  - The expression can be assigned to a variable with the declared type of the method (if the method is not void)
  - There's no expression (if the method is void)

Semantic Analysis

- Parser builds abstract syntax tree
- Now need to extract semantic information and check constraints
  - Can sometimes be done during the parse, but often easier to organize as separate phases
    - And some things can't be done on the fly during the parse, e.g., information about identifiers that are used before they are declared (fields, classes)
- Information stored in symbol tables

Error Recovery

- Common example: What to do when an undeclared identifier is encountered?
  - Only complain once (Why?)
  - Can forge a symbol table entry for it once you've complained so it will be found in the future
  - Assign the forged entry a type of "unknown"
  - "Unknown" is the type of all malformed expressions and is compatible with all other types to avoid redundant error messages

“Predefined” Things

- Many languages have some “predefined” items
- Include code in the compiler to manually create symbol table entries for these when the compiler starts up
  - Rest of compiler generally doesn’t need to know the difference between “predeclared” items and ones found in the program

Types

- Classical roles of types in programming languages
  - Run-time safety
  - Compile-time error detection
  - Improved expressiveness (method or operator overloading, for example)
  - Provide information to optimizer

Type Checking Terminology

Static vs. dynamic typing

- static: checking done prior to execution (e.g. compile-time)
- dynamic: checking during execution

Strong vs. weak typing

- strong: guarantees no illegal operations performed
- weak: can't make guarantees

Caveats:

- Hybrids common
- Inconsistent usage common
- "untyped," "typeless" could mean dynamic or weak

	static	dynamic
strong	Java	Lisp
weak	C	PERL (1-5)

Type Systems

- Base Types
  - Fundamental, atomic types
  - Typical examples: int, double, char
- Compound/Constructed Types
  - Built up from other types (recursively)
  - Constructors include arrays, records/ structs/classes, pointers, enumerations, functions, modules, ...

Type Equivalance

- For base types this is simple
  - Types are the same if they are identical
  - Normally there are well defined rules for coercions between arithmetic types
    - Compiler inserts these automatically or when requested by programmer (casts)

Type Equivalence for Compound Types

- Two basic strategies
  - *Structural equivalence*: two types are the same if they are the same kind of type and their component types are equivalent, recursively (i.e., graphs match)
  - *Name equivalence*: two types are the same only if they have the same name, even if their structures match
- Different language design philosophies

Structural Equivalence

- Structural equivalence says two types are equal iff they have same structure
  - atomic types are tautologically the same structure
  - if type constructors:
    - same constructor
    - recursively, equivalent arguments to constructor
- Ex: atomic types, array types, ML record types
- Implement with recursive implementation of equals, or by canonicalization of types when types created then use pointer equality

Name Equivalence

- Name equivalence says that two types are equal iff they came from the same textual occurrence of a type constructor
  - Ex: class types, C struct types (struct tag name), datatypes in ML
  - special case: type synonyms (e.g. typedef) don't define new types
- Implement with pointer equality assuming appropriate representation of type info

Type Casts

- In most languages, one can explicitly cast an object of one type to another
  - sometimes cast means a conversion (e.g., casts between numeric types)
  - sometimes cast means a change of static type without doing any computation (casts between pointer types or pointer and numeric types)

Type Conversions and Coercions

- In Java, can explicitly convert an value of type double to one of type int
  - can represent as unary operator
  - typecheck, codegen normally
- In Java, can implicitly coerce an value of type int to one of type double
  - compiler must insert unary conversion operators, based on result of type checking

C and Java: type casts

- In C: safety/correctness of casts not checked
  - allows writing low-level code that's type-unsafe
  - more often used to work around limitations in C's static type system
- In Java: downcasts from superclass to subclass include run-time type check to preserve type safety
    - static typechecker allows the cast
    - codegen introduces run-time check
    - Java's main form of dynamic type checking

Coming Attractions

- Semantics checking for MiniJava project
- Then on to code generation...