COMS W4115
Programming Languages and Translators
Lecture 15: Intermediate Code Generation
March 25, 2013

Lecture Outline

1. Intermediate Representations
2. Semantic analysis
3. Purpose of types in programming languages
4. Type systems
5. Typing in programming languages
6. Type inference rules
7. Type conversions

1. Intermediate Representations

- Abstract syntax trees
- Directed acyclic graphs (DAGs)
- Three-address code
  - Addresses
    - Names
    - Constants
    - Compiler-generated temporaries
  - Instructions
    - Three-address code has instructions of various forms for assignments, conditional and unconditional jumps, procedure calls and returns, indexed copy, address and pointer assignments.
  - Common representations for three-address code include records, quadruples, and triples.
- Static single-assignment (SSA) form is an intermediate reprentation that facilitates certain code optimizations. Assignments in SSA are to variables with distinct names. It also uses a special function, called a Ï†-function, to combine two definitions of the same variable arising from two different control-flow paths.

2. Semantic Analysis

- The semantic analyzer uses the syntax tree and information in the symbol table to check the source program for semantic consistency with the language definition.
- It gathers type information for intermediate code generation.
- Type checking is an important part of semantic analysis. During type checking the compiler checks that each operator has compatible operands.
- Uses made of semantic information for a variable x:
  - What kind of value is stored in x?
  - How big is x?
  - Who is responsible for allocating space for x?
  - Who is responsible for initializing x?
  - How long must the value of x be kept?
  - If x is a procedure, what kinds of arguments does it take and what kind of return value does it have?
- Storage layout for local names

3. Purpose of Types in Programming Languages

- Virtually all high-level programming languages associate types with values.
- Types often provide an implicit context for operations. For example, in C the expression `x + y` will use integer addition if x and y are `int`'s, and floating-point addition if x and y are `float`'s.
- Types can catch programming errors at compile time by making sure operators are applied to semantically valid operands. For example, a Java compiler will report an error if x and y are `String`'s in the expression `x * y`.

4. Type Systems

- The type of a construct in a program can be denoted by a type expression.
- A type expression is either a basic type (e.g., `integer`) or a type constructor applied to a type expression (e.g., a function from an integer to an integer).
- A type system is a set of rules for assigning type expressions to the syntactic constructs of a program and for specifying
  - type equivalence (when the types of two values are the same),
  - type compatibility (when a value of a given type can be used in a given context), and

    - type inference (rules that determine the type of a language construct based on how it is used).
- Forms of type equivalence
    - Name equivalence: two types are equivalent iff they have the same name.
    - Structural equivalence: two types are equivalent iff they have the same structure.
    - To test for structural equivalence, a compiler must encode the structure of a type in its representation. A tree (or type graph) is typically used.
- A type checker makes sure that a program obeys the type-compability rules of the language.
- We can think about types in several different ways:
    - Denotational: a type is a set of values called a domain.
    - Constructive: a type is either a primitive type (such as an integer or a character) or a composite type created by applying a type constructor (such as a structure or an array) to simpler types.
    - Abstraction-based: a type is an interface consisting of a set of operations with well-defined and mutually consistent semantics.

6. Typing in Programming Languages

- The type system of a language determines whether type checking can be be performed at compile time (statically) or at run time (dynamically).
- A statically typed language is one in which all constructs of a language can be typed at compile type. C, ML, and Haskell are statically typed.
- A dynamically typed language is one in which some of the constructs of a language can only be typed at run time. Perl, Python, and Lisp are dynamically typed.
- A strongly typed language is one in which the compiler can guarantee that the programs it accepts will run without type errors. ML and Haskell are strongly typed.
- A type-safe language is one in which the only operations that can be performed on data in the language are those sanctioned by the type of the data. [Vijay Saraswat]

7. Type Inference Rules

- Type inference rules specify for each operator the mapping between the types of the operands and the type of the result.
- E.g., result types for `x + y`:

+	**int**	**float**
int	int	float
float	float	float

- Operator and function overloading
- In Java the operator + can mean addition or string concatenation depending on the types of its operands.
- We can choose between two versions of an overloaded function by looking at the types of their arguments.
- Function calls
- Compiler must check that the type of each actual parameter is compatible with the type of the corresponding formal parameter. It must check that the type of the returned value is compatible with the type of the function.
- The *type signature* of a function specifies the types of the formal parameters and the type of the return value.
- Example: `strlen` in C
    - Function prototype in C:

```
unsigned int strlen(const char *s);
```

- Type expression:

```
strlen: const char * â†’ unsigned int
```

- Polymorphic functions
- A polymorphic function allows a function to manipulate data structures regardless of the types of the elements in the data structure
- Example: Fig. 6.28 (p. 391) -- an ML program for the length of a list

8. Type Conversions

- Implicit type conversions
    - In an expression like `f + i` where `f` is a float and `i` is an integer a compiler must first convert the integer to a float before the floating point addition operation is performed. That is, the expression must be transformed into an intermediate representation like

```
t1 = INTTOFLOAT i
t2 = x FADD t1
```

- Explicit type conversions
- In C, explicit type conversions can be forced ("coerced") in an expression using a unary operator called a cast. E.g., `sqrt((double) n)` converts the value of the integer n to a `double` before passing it on to the square root routine `sqrt`.

9. Practice Problems

The following grammar generates programs consisting of a sequence of declarations D followed by a single expression E. Each identifier must be declared before its use.

```
P â†’ D ; E
D â†’ D ; D | T id
T â†’ int | float | T [ num ]
E â†’ num | id | E [ E ] | E + E
```

- Construct type expressions as in Section 6.3.1 (pp. 371-372) for the following programs:

a. int a; int b; a + b

b. float[10][20] a; a[1] + a[2]

- Write pseudcode for a function `sequiv(exp1, exp2)` that will test the structural equivalence of two type expressions `exp1` and `exp2`.
- Show how your function computes `sequiv(array(2, array(2, int)), array(2, array(3, int)))`.

10. Reading

- ALSU, Sects. 6.1-6.4.

---

aho@cs.columbia.edu