编译原理/Compiler
Semantic Analysis

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/11/9

编译原理

References

- http://www.stanford.edu/class/cs143/
  - Partial contents are copied from the slides of Alex Aiken

---

编译原理

Outline

- Symbol Table
- Type Check

编译原理

Abstract Syntax Trees

- Separate AST construction from semantic checking phase
- Traverse the AST and perform semantic checks (or other actions) only after the tree has been built and its structure is stable
- This approach is less error-prone
  - It is better when efficiency is not a critical issue

编译原理

Incorrect Programs

- Lexically and syntactically correct programs may still contain other errors!
- Lexical and syntax analysis are not powerful enough to ensure the correct usage of variables, objects, functions, statements, etc.

编译原理

Incorrect Programs

- Example 1: lexical analysis does not distinguish between different variable or function identifiers (it returns the same token for all identifiers)

```
int a;        int a;
a = 1;        b = 1;
```

- Example 2: syntax analysis does not correlate the declarations with the uses of variables in the program:

```
int a;
a = 1;        a = 1;
```

- Example 3: syntax analysis does not correlate the types from the declarations with the uses of variables:

```
int a;        int a;
a = 1;        a = 1.0;
```

编译原理

Goals of Semantic Analysis

- Semantic analysis = ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements)
- Examples of semantic rules:
  - Variables must be declared before being used
  - A variable should not be declared multiple times in the same scope
  - In an assignment statement, the variable and the assigned expression must have the same type
  - The condition of an if-statement must have type boolean
- Some categories of rules:
  - Semantic rules regarding types
  - Semantic rules regarding scopes

编译原理

Type Information

- Type information = describes what kind of values correspond to different constructs: variables, statements, expressions, functions
- variables: int a; integer
- expressions: (a+1) == 2 boolean
- statements: a = 1.0 floating-point
- functions: int pow(int n, int m) int x int → int

编译原理

Type Checking (cont.)

- Type checking = set of rules that ensure the type consistency of different constructs in the program
- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

编译原理

Type Checking (cont.)

- More examples:
  - For each assignment statement, the type of the updated variable must match the type of the expression being assigned
  - For each call statement foo(v_1, …, v_n),
    - the type of each actual argument v_i must match the type of the corresponding formal argument f_i from the declaration of function foo
    - the type of the return value must match the return type from the declaration of the function

编译原理

Scope Information

- Scope information = characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier
  - Example identifiers: variables, functions, objects, labels
- Lexical scope = textual region in the program
  - Statement block
  - Formal argument list
  - Object body
  - Function or method body
  - Module body
  - Whole program (multiple modules)
- Scope of an identifier: the lexical scope in which it is valid

编译原理

Scope of variables in statement blocks

```
{ int a;
…
    { int b;
    }
…
}
```

b

a

- In C:
  - Scope of file static variables: current file
  - Scope of external variables: whole program
  - Scope of automatic variables, formal parameters, and function static variables: the function

Scope of formal arguments of functions/methods

```
int factorial(int n) {
…
}
```

n

- Scope of labels

```
void f() {
… goto l; …
l: a =1;
… goto l; …
}
```

l

Scope of object fields and methods

```
class A {
private int x;
public void g() { x=1; }
…
}
class B extends A {
…
public int h() { g(); }
…
}
```

x

g

Semantic Rules for Scopes

- Main rules regarding scopes:
  - Rule 1: Use an identifier only if defined in enclosing scope
  - Rule 2: Do not declare identifiers of the same kind with identical names more than once in the same lexical scope
- Can declare identifiers with the same name with identical or overlapping lexical scopes if they are of different kinds

```
class X {
    int X;
    void X(int X){
        X: for(;;)
        break X;
    }
}
int X(int X){
    int X;
    goto X;
    { int X;
    X: X = 1; }
}
```

Not Recommended!

SYMBOL TABLE

Symbol Tables

- Semantic checks refer to properties of identifiers in the program -- their scope or type
- Need an environment to store the information about identifiers = symbol table
- Each entry in the symbol table contains
  - the name of an identifier
  - additional information: its kind, its type, if it is constant, …

Symbol Tables(Example)

NAME	KIND	TYPE	ATTRIBUTES
foo	fun	int * int → bool	extern
m	arg	int	
n	arg	int	const
tmp	var	bool	const

编译原理

Scope Information

- How to capture the scope information in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use a similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

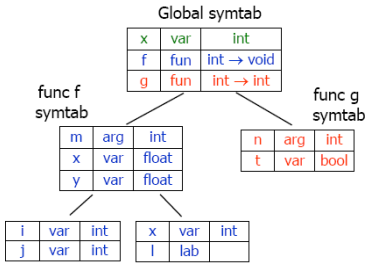
编译原理

Example

```
int x;

void f(int m) {
   float x, y;
   ...
  { int i, j; ...; }
  { int x; l: ...; }
}

int g(int n) {
   bool t;
   ...;
}
```



编译原理

Identifiers with Same Name

- The hierarchical structure of symbol tables automatically solves the problem of resolving name collisions (identifiers with the same name and overlapping scopes)
- To find which is the declaration of an identifier that is active at a program point:
  - Start from the current scope
  - Go up in the hierarchy until you find an identifier with the same name

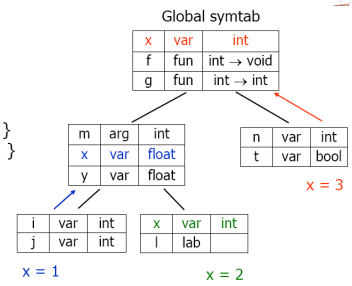
编译原理

Example

```
int x;

void f(int m) {
   float x, y;
   ...
  { int i, j; x = 1; }
  { int x; l: x = 2; }
}

int g(int n) {
   bool t;
   x = 3;
}
```



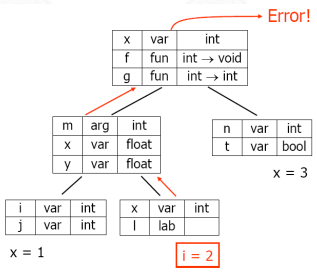
编译原理

Catching Semantic Errors

```
int x;

void f(int m) {
   float x, y;
   ...
  { int i, j; x = 1; }
  { int x; l: i = 2; }
}

int g(int n) {
   bool t;
   x = 3;
}
```



编译原理

Symbol Table Operations

- Two operations:
  - insert
  - lookup
- Each entry can be implemented as a record. Records can have different formats (Variant records in Pascal).

编译原理

Symbol Table Operations

- Cannot build symbol tables during lexical analysis
  - hierarchy of scopes encoded in the syntax
- Build the symbol tables:
  - while parsing, using the semantic actions
  - After the AST is constructed

编译原理

Storing characters

- Method 1: A fixed size space within each entry large enough to hold the largest possible name.
  - Most names will be much shorter than this so there will be a lot of wasted storage
- Method 2: Store all symbols in one large separate array. Each symbol is terminated with an end of symbol mark (EOS).
  - Each symbol table record contains a pointer to the first character of the symbol.
- Method n: modern languages (e.g. Java, C++ std components) has efficient DS, e.g. string or vector

编译原理

Symbol Table Data Structure

- One Linear list:
  - Easy to implement
  - Search time will be very long if source has many symbols.
- Simple implementation = array
  - One entry per symbol
  - Disadvantage:
    - table has fixed size
    - need to know in advance the number of entries
- Dynamic structure = list
  - Can grow dynamically during compilation
  - need to scan half the list on average

编译原理

Symbol Table Data Structure

Hash table:

- Run the symbol name through a hash function to create an index in a table.
- If some other symbol has already claimed the space then rehash with another hash function to get another index, etc.
- Hash Table must be large enough to accommodate largest number of symbols.

编译原理

Symbol Table Data Structure

- Open hash:
  - Store the entries in a number of linear lists (called Buckets).
  - Use a hash function on the symbol name to determine which lists to use.
  - A good hash function will spread the symbols across the buckets, so each linear list will be short.

编译原理

Forward References

- Forward references = use an identifier within the scope of its declaration, but before it is declared
  - Forward declaration
- Any compiler phase that uses the information from the symbol table must be performed after the table is constructed
  - Cannot type-check and build symbol table at the same time

```
Example:
class A {
    int m() { return n(); }
    int n() { return 1; }
}
```

编译原理

Summary

- Semantic checks ensure the correct usage of variables, objects, expressions, statements, functions, and labels in the program
- Scope semantic checks ensure that identifiers are correctly used within the scope of their declaration
- Type semantic checks ensures the type consistency of various constructs in the program
- Symbol tables: a data structure for storing information about symbols in the program
  - Used in semantic analysis and subsequent compiler stages

编译原理

TYPE CHECKING

编译原理

What Are Types?

- Types describe the values computed during the execution of the program
- Essentially, types are predicate on values, e.g., "int x" in Java means "$x \in [-2^{31}, 2^{31})$"
- Type errors: improper, type-inconsistent operations during program execution
- Type-safety: absence of type errors

编译原理

How to Ensure Type-Safety

- Bind (assign) types, then check types
- Type binding: defines type of constructs in the program (e.g., variables, functions)
  - Can be either explicit (int x) or implicit (x = 1)
  - Type consistency (safety) = correctness with respect to the type bindings
- Type checking: determine if the program correctly uses the type bindings
  - Consists of a set of type-checking rules

编译原理

Type Checking

- Type checking: static semantic checks to enforce the type safety of the program
- Examples:
  - Unary and binary operators (e.g., +, ==, [ ]) must receive operands of the proper type
  - Functions must be invoked with the right number and type of arguments
  - Return statements must agree with the return type
  - In assignments, assigned value must be compatible with type of variable on LHS.
  - Class members accessed appropriately

编译原理

Static vs. Dynamic Typing

- Static and dynamic typing refer to type definitions (i.e., bindings of types to variables, expressions, etc.)
  - Statically typed language: types are defined and checked at compile-time, and do not change during the execution of the program
    - E.g., C, Java, Pascal
  - Dynamically typed language: types defined and checked at run-time, during program execution
    - E.g., Lisp, Smalltalk

编译原理

Strong vs. Weak Typing

- Strong and weak typing refer to how much type consistency is enforced
  - Strongly typed languages: guarantees that accepted programs are type-safe
  - Weakly typed languages: allow programs that contain type errors
- Can achieve strong typing using either static or dynamic typing

编译原理

Soundness

- Soundness type systems: can statically ensure that the program is type-safe.
- Soundness implies strong typing.
- Static type safety requires a conservative approximation of the values that may occur during all possible executions
  - May reject type-safe programs
  - Need to be expressive: reject as few type-safe programs as possible

编译原理

Why Static Checking?

- Efficient code
  - Dynamic checks slow down the program
- Guarantees that all executions will be safe
  - Dynamic checking gives safety guarantees only for some execution of the program
- But is conservative for sound systems
  - Needs to be expressive: reject few type-safe programs

编译原理

Type Systems

- Type is predicate on value
  - Type expressions: describe the possible types in the program: int, string, array[], Object, etc.
  - Type system: defines types for language constructs (e.g., expressions, statements)

编译原理

Type Systems

- The design of a type checker for a language is based on
  - the syntactic constructs in the language
  - the notion of types
  - the rules of assigning types to language constructs
- In C and Pascal
  - arithmetic
  - pointer operator
- Each expression have a type associated it.
- Types have structure.

编译原理

Kinds of Types

- Basic Type:
  - boolean, char, int, real, enum
- Constructed Type:
  - arrays, records, sets, pointers, functions

编译原理

Type Expressions

- Basic type
- Type name
- Type Constructor

编译原理

Basic type

- boolean, char, int, real, enum
- void
  - the absence of a value
  - allows statements to be checked
- type_error
  - Special basic type
  - Will signal an error during type checking

编译原理

Type Constructors

- Can be applied to type expressions so that a new type expression generated
- Typical type constructors
  - Arrays: array(I, T)
  - Products: T1×T2
  - Structures: record((id$_1$×T1)×(id$_2$×T2) × …)
  - Pointers: pointer(T)
  - Functions: T1× … ×Tn → T

编译原理

Type Constructors

- Various kinds of array types in different
  - array(T, S)
  - T : element type
  - S : index set, often a range of integers
- programming languages
  - array(T) : arrays without bounds
    - C, Java: T [ ], Modula-3: array of T
  - array(T, S) : array with size
    - C: T[S], Modula-3: array[S] of T
    - May be indexed 0..S-1
  - array(T,L,U) : array with upper/lower bounds
    - Pascal: array[L .. U] of T
  - array(T, S1, …, Sn) : multi-dimensional arrays
    - FORTRAN: T(L1,…, Ln)

编译原理

Structures(Records)

```
Structures in C:
    struct { int a; float b; }
Records in Pascal:
  record a: integer; b: real;
end
```

```
struct list {
    struct list* next ;
    int value ;
}
```

- Field names should be part of the type constructor
- Sometime it is more convenient to represent record as the following:
  - record((id$_1$×T1)×(id$_2$×T2) × …)

编译原理

Type Constructor

- Pointers:
  - pointer(T)
  - pointer to an object of type T

C pointers:
T* (e.g., int *x;)
Pascal pointers:
^T (e.g., x: ^integer;)

- Functions:
  - Mapping elements of domain to range
  - T1× … ×Tn→T

C functions :
int f(float x, floaty)

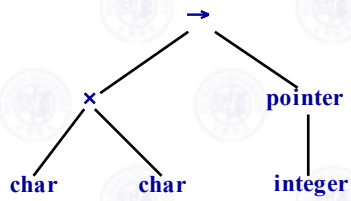
编译原理

Type names

- Type expressions may be named
- A type name is a type expression

编译原理

Type Expressions



编译原理

Type Systems

- A Type System is
  - a collection of rules for
  - assigning type expressions to
  - the various parts of a program

编译原理

Type-Checking

- (Option) first build the AST, then implement type-checking by recursive traversal of the AST nodes:

```
class Add extends Expr {
   Type typeCheck (Symtab s) {
        Type t1 = e1.typeCheck(s),
        t2 = e2.typeCheck(s);
        if (t1 == Int && t2 == Int) return Int;
        else throw new TypeCheckError("+");
   }
}
```


编译原理

Static and Dynamic Checking of Types

- Static:
  - Checking done by a compiler
- Dynamic:
  - Checking done when the target program runs
- A sound type system eliminates the need for dynamic checking for type errors.

编译原理

Error Recovery

- The compiler must report the nature and location of error.
- It is desirable for the type checker to recover from errors.

A Simple Language

- P → D ; E
- D → D ; D | id : T
- T → char | integer | array [num] of T | *T
- E → literal | num | id | E mod E | E [E] | *E

- key: integer;
- key mod 1999

Semantic Actions

- id : char
- addtype(id.entry, char)

- id : integer
- addtype(id.entry, integer)

Semantic Actions

- T → array [num] of T1

- T.type = array(1..num.val, T1.type)

- T → *T1
- T.type = pointer(T1.type)

Type Checking of Expressions

- E → literal
- E.type = char

- E → num
- E.type = integer

- E → id
- E.type = lookup(id.entry)

Type Checking of Expressions

- E → E1 mod E2
- E.type = (E1.type==integer && E2.type==integer) ? integer : type_error

Type Checking of Expressions

- E → E1 [E2]
- E.type = (E2.type == integer && E1.type == array(s, t)) ? t: type_error}

Type Checking of Expressions

- E → *E1
- E.type = (E1.type == pointer(t)) ? t : type_error

Type Checking of Functions

- T → T1 '→' T2
- T.type = T1.type → T2.type
- E → E1(E2)
- E.type = (E2.type == s && E1.type == s → t) ? t : type_error

Names for Type Expressions

- typedef cell *link;
- link next;
- link last;
- cell *p;
- cell *q, *r;
- next, last link
- p, q, r pointer(cell)
-

Names for Type Expressions

- Name Equivalence
  - Views each type name as a distinct type
- Structural Equivalence
  - Names are replaced by the type expressions they defined

Structural Equivalence of Type Expressions

- Two expressions are
  - Either the same basic type
  - Or are formed by applying the same constructor to structurally equivalent types

Algorithm

```
int sequiv(s, t) {
        if ( s and t are the same basic type ) return 1 ;
        else if ( s == array(s1, s2) && t == array(t1, t2) )
                return sequiv(s1, t1) && sequiv(s2, t2)
        else if ( s == s1×s2 && t == t1 × t2 )
                return sequiv(s1, t1) && sequiv(s2, t2)
        else if ( s == pointer(s1) && t == pointer(t1) )
                return sequiv(s1, t1)
        else if ( s == s1→s2 && t == t1 → t2 )
                return sequiv(s1, t1) && sequiv(s2, t2)
        return 0;
}
```

Names for Type Expressions

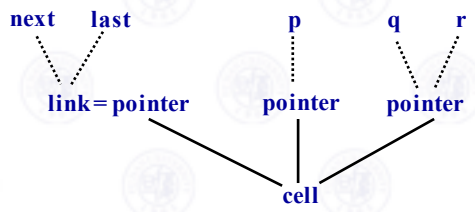
- typedef cell *link;
- link next ;
- link last ;
- cell *p ;
- cell *q, *r ;

- next, last link
- p, q, r pointer(cell)
-

Type Variables

- Variables representing type expressions allow us to talk about unkown types.

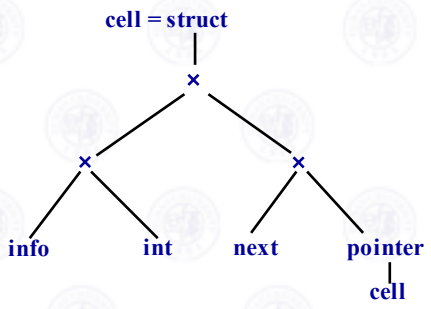
Names for Type Expressions



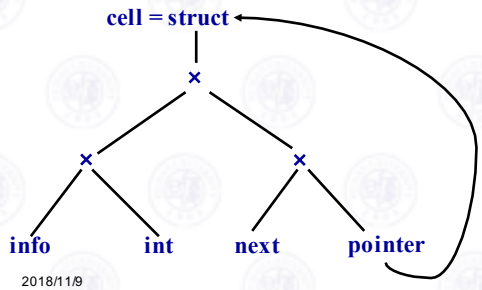
Cycles in Representations of Types

```
typedef cell *link ;
typedef struct {
        int info ;
        link next ;
} cell ;
```


Cycles in Representations of Types



Cycles in Representations of Types



编译原理

Type Conversion

- X + I
  - X real, I int
  - Insert inttoreal before I
  - X I inttoreal real+
- Implicit type conversion
  - Coercions
    - No information lost
    - int to double, not vice-versa
- Explicit type casting

编译原理

Type Conversion

Production	Semantic rule
E → **num**	E.type = integer
E → **num.num**	E.type = real
E → **id**	E.type = lookup(**id**.entry)
E → E_1 **op** E_2	E.type = if (E_1.type = integer && E_2.type = integer) then integer else if (E_1.type = integer && E_2.type = real) then real else if (E_1.type = real && E_2.type = integer) then real else if (E_1.type = real && E_2.type = real) then real else type_error

编译原理

Simple Overloading

- Overloaded symbol has different meanings depending on its arguments
  - 1.0+2.0
  - 1 + 2
- Resolution
  - Determine a unique meaning for an occurrence of an overloading symbol
    - Well known as operator identification

编译原理

Narrowing the Set of Possible Types

- Given a unique type from the context, we narrow down the type choices for each subexpression.

编译原理

General Overloading

- The meaning of an overloading symbol is depending on its context
- An example from Ada, operator "*"
  - A pair of integers to an integer
  - A pair of integers to a complex
  - A pair of complexes to a complex
  - 2*(3*5)
    - (3*5) must be integer
  - (3*5)*z
    - (3*5) must be complex

编译原理

General Overloading

- E → E1(E2)
- E.type = {(E2.types == s && E1.types == s → t) ? t : type_error}
- E.types = { t | there exists an s in E2.types such that (s → t) is in E1.types }

编译原理

General Overloading