Compiler Technology of Programming Languages

Chapter 9
Semantic Analysis

Prof. Farn Wang

Outlines of Chapter 9

- Semantic analysis for control structures
  - Reachability and Termination Analysis
  - If statements
  - Switch and Case
  - While, Do and Repeat Loops
  - For loops
  - Break, Continue, Return and GoTo
  - Exception Handling

Select statements

Iterative statements

Departure from normal path

- Semantic analysis of Calls

Semantic Analysis for Control Structures

- Three aspects of semantic analysis
  - Type Correctness
    - Chapter 8 covered this type of analysis
  - Reachability and termination
    - determines if a construct will ever be executed and will terminate normally
    - required by Java and C#
  - Exception Handling
    - Constructs may throw exceptions rather than terminate normally.

Semantic Analysis for Control Structures

- Three type of visitors
  - Semantics visitor

    checks type rules
  - Reachability visitor

    Analyzes control structures for reachability and proper termination

    Required by Java and C#
    Optional for C and C++
  - Throw visitor

    Computes the *ThrowsSet* field, which records exceptions that may be thrown

Exceptions

- Anomalous or exceptional conditions during execution that require special processing.
- Hardware exception handling/traps
  - Traps are internal interrupts, if no registered exception handler, traps will cause core dump – program termination.
  - Example: misaligned access, segmentation faults, floating point underflow.
  - OS provides signal supports:
    - E.g. SIGSEGV, SIGILL, SIGBUS, SIGFPE
- Software exception handling
  - Offer programmers a chance to handle it gracefully.
  - Many modern PLs (C++,C#, Java, ML, Objective-C, Python, Ruby, Scala, .NET languages) support exception handling.

Rules for Java Exceptions

- Catch or Declare Rule
  - All checked exceptions that may occur in a procedure must be caught by that procedure or are listed in that procedure's throws clause
  - All checked exceptions that may be propagated to a caller from a procedure must be listed in that procedure's throws clause.
  - All checked exceptions that are listed in a procedure's throws clause must be propagated to a caller from that procedure in some situations.

Checked vs. Unchecked Exceptions

- Checked exceptions:
  - if you know how to deal with using some alternative code.
  - must be dealt with in a try/catch block or by declaring a “throws” clause in a method.
- Unchecked exceptions:
  - you don’t know how to deal with, and they should never happen
  - if they do, it is a bug.
  - If they happen, let them come up to the surface and displayed to the user.
  - normally runtime exceptions like NullPointerException.

Programmers usually tend to ignore that responsibility.

Basic Try-Throw-Catch

```
try { if (num==0)
        throw new exception("error! …");
      ….}
  catch (exception e)
  {
        String message = e.getMessage();
        System.out.println(message);
        System.exit(0);
  }
```

Exception handler

Throwing Exceptions in a Method

```
Public void safedivide(int top, into bottom)
throws divbyzeroException
{
        if (bottom == 0)
                throw new divbyzeroException();
        return top/bottom;
}
```

```
Public class divbyzeroException extends Exception
{
        constructor…
        access methods …
}
```

Catching Exceptions in the Caller

```
Public void calldivide() {
    try {
        int result = safedivide(2,0);
        ….}
    catch (divbyzeroException, e);
    // do something with the exception
```

Propagate Exceptions

```
Public void calldivide()
throws divbyzeroException
        {
                int result = safedivide(2,0);
                ….
        }
```

Uncaught exceptions must propagate up the call stack

Reachability and Termination Analysis

- Java requires *unreachable* statements be identified.

```
…;
return;
a=a+1;
…
```

- In the general case, to tell whether a statement is unreachable is undecidable.

So the reachability analysis is usually *conservative.*

- *answering unreachable or don't know*

conservative here means "err on one side"

Example

1) for (i=0; c < n; i++) { c = f(i); }

Will this loop ever terminate?

*2) If (**a** > **b**) {…}*

*else if (**a** > **c**) { S1 }*

Will S1 be reachable?

*What if when **a** <= **b** then **a** is always <= **c** ?*

Example

1) for (i=0; c < n; i++) { c = f(i); }

Will this loop ever terminate?

Conservative: TerminateNormally

*2) If (**a** > **b**) {…}*

*else if (**a** > **c**) { S1 }*

Will S1 be reachable?

*What if when **a** <= **b** then **a** is always <= **c** ?*

Conservative: isReachable

Conservative Analysis in Compilers

Example 1

isReachable ➔ may be reachable

Example 2:

terminateNormally ➔ may terminate normally

Example 3:

variable a is initialized ➔ may be initialized

Example 4: alias/heap/pointer analysis

*p may alias with *q

p may alias with variable a,b[3],(*(c+2)+5)

Termination analysis

A statement terminates normally if execution continues to the next statement.

According to this criterion,

- statements such as break, continue, return, etc. do not terminate normally.
- An infinite loop, such as for (;;) {}, does not terminate normally, either.

Rules for *isReachable* and *TerminateNormally*

- Each STMT and STMT_list (nonterminals) AST nodes will include two attributes:

  isReachable and ***TerminateNormally***

- If a *statement list* is reachable, it is also true for its first statement
- If TerminateNormally is false for the last statement in a statement list, it is also false for the whole list.
- The statement list that comprises the body of a method, constructor or a static initializer is always considered reachable.

Rules for *isReachable* and *TerminateNormally*

- A local variable declaration or an expression statement always have TerminateNormally true.
- A null statement never generates an error message if it is not reachable. The ***isReachable*** value is propagated to the successor.
- A statement is reachable if and only if its predecessor terminate normally.

True meaning:

isReachable → ***is possibly reachable***

TerminateNormally → ***may terminate normally***

Example

```
Void example ()
{   int v;
    v++; return; ; v=10; v=20; }
```

- the method body is reachable
- the declaration is reachable
- the declaration and the increment terminate normally
- The return is reachable, but does not terminate normally
- the null statement is not reachable
- V=10 is not reachable, and generate an error msg

Function

- If the STMT list of a function body is terminate abnormally, that means it does not a return. So the needs to issue an error message.

IF statements

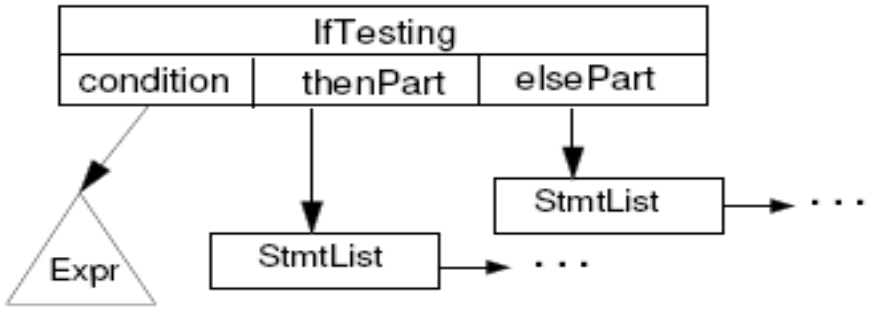

Figure 9.2: Abstract Syntax Tree for an If Statement

1) Check whether the predicate (Expr) has the boolean or errorType type.
2) Type check for children (then and else)
3) Reachability check and collect exceptions from each child.

What if the predicate is known (T or F)?

Example of Checking an IF statement

If (b) a = 1;
else a = 2;

1)Check that the predicate b produces a boolean value
2)Two children (then and else) are type checked.
3)Since the two statements terminate normally, so does the whole IF statement.

For Loops

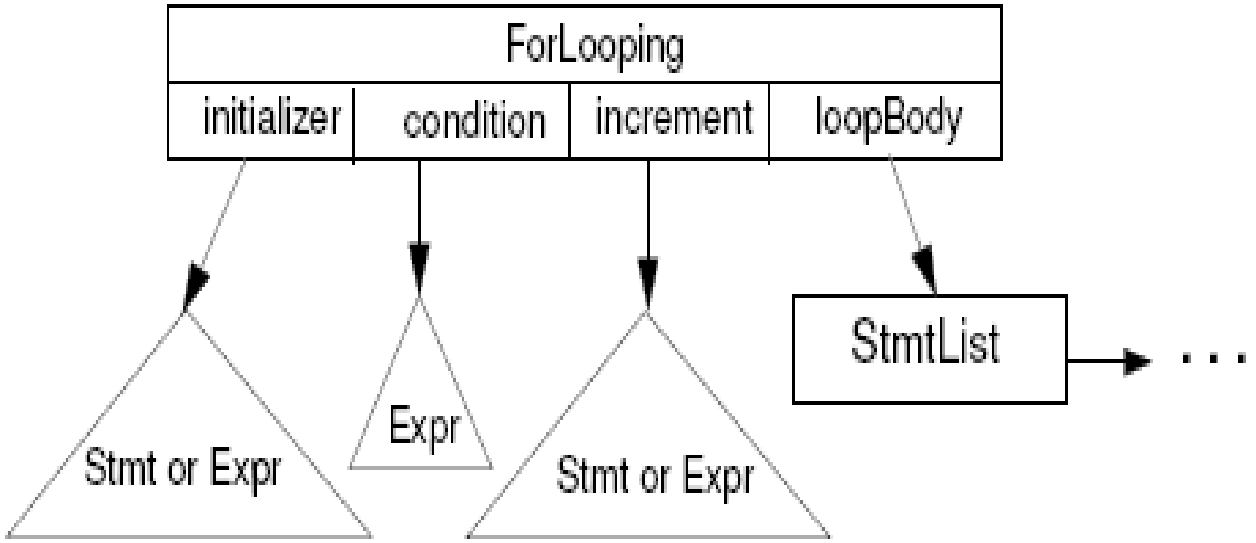

Figure 9.9: Abstract Syntax Tree for a For Loop

Semantic Checking of **For**

- In C++, C# and Java, a new index variable local to the **for** loop may be declared.

  e.g. for (int n = 1; n < 10; n++)
- A null condition and a constant-true predicate make the loop non-terminating
- Gather all the ThrowsSet from all children

Exception Checks

- Exceptions may be thrown explicitly or implicitly
- Thrown exceptions may be propagated several times and then finally get caught by a handler.
- Java exceptions are typed: all exceptions are subclasses of the Throwable class.
- Two kinds of exceptions: *checked* and *unchecked*.
  - A checked exception must be caught by an enclosing **try** statement or is listed in the **throws** clause of the enclosing method.
  - exceptions that are exempt from the *Catch or Declare* rule are called *unchecked exceptions*.

Exception Checks

```
class ExitComputation extends Exception { . . . }
try { . . .
        if (a > b) throw new ExitComputation();
        if (v < 0.0) throw new ArithemticException();
        else a =Math.sqrt(v)
}
catch (ExitComputation e) { print(e); return(0); }
catch (ABCException e) { print(e); return(5); }
finally { a := b + c; }
```

Exception Checks

- An unchecked exception (i.e. a *RuntimeException* or an Error) may optionally be handled in a try statement.
  - If not caught by the program, it will be passed to the Java runtime system, and the program will terminate.
  - For example, the ArithemeticException, a subclass of *RuntimeException*, is an unchecked exception.
- For the catch clause, we must check
  - if indeed an exception is thrown
  - if the exception subsumes (i.e. hide) a later exception in the list. (the compiler may either warn, or reverse the order)
- For the throw statement
  - Check if indeed the parameter is an exception object

Exception Checks

- Error Case 1

```
int mm() throws A, B {
        try {     throw new E;
        } catch (C e) { ... }
        catch (D e) { ... }
}
```

- Error Case 2

```
int mm() throws A, B {
    try {
      ... throw new A;
      ... throw new B;
      ... throw new C;
      // never throw new D;
    } catch (C e) { ... }
    catch (D e) { ... } // this D handler is redundant
 }
```