

CS 403: Semantic Analysis, continued

Stefan D. Bruda

Winter 2015

REACHABILITY AND TERMINATION



- Both reachability and termination are undecidable!
 - A compiler typically performs a **conservative analysis** (will not identify all the errors)
- Analysis based on the flags *isReachable* and *terminatesNormally* attached to each node in the AST and set according to the following rules:
 - If *isReachable* is true for a statement list then it is also true for the first statement in the list
 - If *terminatesNormally* is false for the last statement of a list then it is false for the whole statement list
 - *isReachable* is always true for the body of a method, constructor, or static initializer
 - *terminatesNormally* is always true for a variable declaration or expression statement (assignment, function call, heap allocation, increment, decrement)
 - An empty statement list with *isReachable* set to false will never generate any error message; its *isReachable* is also propagated to its successor
 - *isReachable* for some statement is set to the value of *terminatesNormally* for the preceding statement (if any)
 - A return statement never terminates normally
 - Control structures may or may not terminate normally (discussed later)

CS 403: Semantic Analysis, continued (S. D. Bruda)

Winter 2015 1 / 8

CONTROL STRUCTURES



- **If statements** are valid iff the condition has type Boolean, and the “then” and “else” branches (if any) are valid (**recursively**)
 - *isReachable* is set to true for both the “then” and the “else” branch
 - *terminatesNormally* is set to the **disjunction** between the flags of the “then” and the “else” branches
- **While loops** are valid iff their condition has type Boolean and their bodies are valid (**recursively**)
 - If the condition is constant and false then the body is marked as unreachable
 - If the condition is constant and true then the loop is marked as terminating abnormally, **unless** a reachable `break` statement exists in the body (case in which the loop terminates normally)
 - If the condition is not constant then the loop is marked as terminating normally
- **Do-while and Repeat loops** are valid in the same sense as the while loop
 - The body of the loop is always reachable
 - If the condition is constant and true then the loop is marked as terminating abnormally, **unless** a reachable `break` statement exists in the body (case in which the loop terminates normally)
 - Otherwise *terminatesNormally* is initially set to false, but it can become true if the body terminates normally

CONTROL STRUCTURES (CONT'D)



- **C-like For loops** are valid iff their initializer and increment are valid statements (**recursively**), the condition has type Boolean or is empty, and the body is valid (**recursively**)
 - However, in a for loop **all the checks above are done in a new scope**
 - Reachability and termination are done similarly with the while loop
- **Pascal-like For loops** are more restrictive and have the following form:

```
for id := initialVal to finalVal do loopBody
```

 - This for statement is valid iff `id` is a scalar type (integer or enumeration), `initialVal` and `finalVal` are valid expressions and have the same type as `id`, and `loopBody` is valid (**recursively**)
 - `id` is also made constant during the analysis of the body since the body is not allowed to change it
- **Continue statements** are valid iff they appear inside an iterative control structure (while, for, etc.)
 - Such verification is dependent on the construction of the AST
 - May require following the parent links of the statement (if available) or suitable labeling of AST nodes during parsing

CS 403: Semantic Analysis, continued (S. D. Bruda)

Winter 2015 2 / 8

CS 403: Semantic Analysis, continued (S. D. Bruda)

Winter 2015 3 / 8



- **Break statements** are handled just like `continue`, except that they are also allowed inside `switch` statements
- **Return statements** are verified as follows:
 - They must appear inside a method or function (determined following parent links or via suitable labeling during parsing – **also useful for determining the return type of that function**)
 - Their argument must have the same type as the return type of the enclosing function
 - Except that they have no argument for `void` functions
 - Return statements are always terminating abnormally
- **Goto statements** depend on labels, which should be available in the AST
 - Usually processed in two steps, one for constructing the list of labels and the next for verifying that the goto statement points to a valid label
 - Label validity depends on the language (local to a function, global, no goto inside an iterative construct from the outside, etc.)



- **Switch statements** are valid iff:
 - The control expression and the case labels must be type checked (scalar and also consistent with each other)
 - Each case label must be constant and no two labels must have the same value
 - At most one default label must be present
 - All the statements in the body must be valid (**recursively**)
 - *terminatesNormally* is first set to false, and becomes true if either of the following is true:
 - the switch body is empty
 - the last case group terminates normally
 - any case group contains a reachable break



- Throw statements must be type checked
 - They must be used to compile a list of exceptions being thrown (see below)
- The body of a try statement must be valid (**recursively**)
 - All the statements are marked as reachable
 - A try terminates normally whenever some catch statement terminates normally and also the default (if any) catch does the same
- Catch statements must have a valid body (**recursively**) and:
 - They introduce a new identifier (in a new scope) which must be inserted into the symbol table
 - A list of all the exceptions being caught should be assembled
 - This list is used to determine which exceptions are caught and which are not
 - The uncaught exceptions must be caught by outer catch statements or declared as being thrown out of the function
 - Uncaught exceptions must be propagated throughout the AST



- We assume an object-oriented language with inheritance and overloading (if these are not present then things are simpler!)
- The first step of analyzing a call is to determine which method definition to use
 - If we have a method call then the “right” call is the nearest in the inheritance hierarchy that has the right parameter types
 - If the method has no qualifier then we examine the current class and all its superclasses
 - If the method has a qualifier that evaluates to an object of type T then we examine the type T and all its superclasses
 - We then gather all the methods from the candidates that match the name + visibility (public, private, protected) + parameter type
 - Actual arguments must be **bindable** to the formal parameters = type checking
 - If we still have more than one candidate, then we choose the most specific
 - We prefer the methods lower in the inheritance hierarchy
 - We prefer methods with arguments lower in the inheritance hierarchy
 - In all a method definition D is **more specific** than definition D' if the class of D is bindable to the class of D' and each parameter of D is bindable to the respective parameter of D'



- If after the process above we end up with one definition then that is the one to call, else we signal an error
- Additional checking:
 - Methods qualified by a class name must be static
 - A call to a void method must not be part of an expression
 - A call to a non-void method must have the return type checked against the rest of the expression
- Constructors are called indirectly but otherwise are checked using the same procedure
- Some languages allow functions to be defined inside functions (Python, ML, etc.)
 - Again same procedure applies
 - Scoping must also be considered (according to the symbol table rules)