CS 403: Type Checking

Stefan D. Bruda

Winter 2015

Semantic Analysis

- Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination – now we move to check whether they form a sensible set of instructions in the programming language $\rightarrow$ semantic analysis
  - Any noun phrase followed by some verb phrase makes a syntactically correct English sentence, but a semantically correct one
    - has subjectverb agreement
    - has proper use of gender
    - the components go together to express an idea that makes sense
- For a program to be semantically valid:
  - all variables, functions, classes, etc. must be properly defined
  - expressions and variables must be used in ways that respect the type system
  - access control must be respected
  - etc.
- Note however that a valid program is not necessariy correct

```
int Fibonacci(int n) {
    if (n <= 1) return 0;
    return Fibonacci(n - 1) + Fibonacci(n - 2); }
int main() { Print(Fibonacci(40)); }
```

Semantic Analysis

- Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination – now we move to check whether they form a sensible set of instructions in the programming language → semantic analysis
  - Any noun phrase followed by some verb phrase makes a syntactically correct English sentence, but a semantically correct one
    - has subjectverb agreement
    - has proper use of gender
    - the components go together to express an idea that makes sense
- For a program to be semantically valid:
  - all variables, functions, classes, etc. must be properly defined
  - expressions and variables must be used in ways that respect the type system
  - access control must be respected
  - etc.
- Note however that a valid program is not necessariy correct

```
int Fibonacci(int n) {
    if (n <= 1) return 0; // should be return 1; !
    return Fibonacci(n - 1) + Fibonacci(n - 2); }
int main() { Print(Fibonacci(40)); }
```

  - Valid but not correct!

Semantic Analysis (cont'd)

- Reject the largest number of incorrect programs
- Accept the largest number of correct programs

Semantic Analysis (cont'd)

- Reject the largest number of incorrect programs
- Accept the largest number of correct programs
- Do so quickly!

http://xkcd.com/303/

Semantic Analysis (cont'd)

- Reject the largest number of incorrect programs
- Accept the largest number of correct programs
- Do so quickly!

http://xkcd.com/303/

- Some semantic analysis done while parsing (syntax directed translation)
  - Some languages specifically designed for exclusive syntax directed translation (one-pass compilers)
  - Other languages require repeat traversals of the AST after parsing
- Several components of semantic analysis:
  - Type and scope checking
  - Other semantic rules (language dependent)

Types and Declarations

- A type is a set of values and a set of operations operating on those values
- Three categories of types in most programming languages:
  - Base types (int, float, double, char, bool, etc.) $\rightarrow$ primitive types provided directly by the underlying hardware
  - Compound types (enums, arrays, structs, classes, etc.) $\rightarrow$ types are constructed as aggregations of the base types
  - Complex types (lists, stacks, queues, trees, heaps, tables, etc) $\rightarrow$ abstract data types, may or may not exist in a language
- In many languages the programmer must first establish the name, type, and lifetime of a data object (variable, function, etc.) through declarations
  - Most type systems rely on declarations
  - Notable exceptions: functional languages that do not require declarations but work hard to infer the data types of variables from the code

Type Checking

- The bulk of semantic analysis = the process of verifying that each operation respects the type system of the language
  - Generally means that all operands in any expression are of appropriate types and number
  - Sometimes the rules are defined by other parts of the code (e.g., function prototypes), and sometimes such rules are a part of the language itself (e.g., "both operands of a binary arithmetic operation must be of the same type")
- Type checking can be done compilation, during execution, or across both
  - A language is considered strongly typed if each and every type error is detected during compilation
  - Static type checking is done at compile-time
    - The information needed is obtained via declarations and stored in a master symbol table
    - The types involved in each operation are then checked
  - Dynamic type checking is implemented by including type information for each data location at run time

The Symbol Table Interface

- The symbol table is used to keep track of which declaration is in effect upon encountering a reference to an id
  - Used in both type and scope checking, so it must keep track of scopes as well as declarations
- A suitable interface therefore contains the following functions
  - enterSymbol(*name*, *type*) $\rightarrow$ adds the id *name* in the symbol table (current scope) with type *type*
  - retrieveSymbol(*name*) $\rightarrow$ returns the currently valid entry in the symbol table for *name* or a null pointer if no such entry exists

The Symbol Table Interface

- The symbol table is used to keep track of which declaration is in effect upon encountering a reference to an id
  - Used in both type and scope checking, so it must keep track of scopes as well as declarations
- A suitable interface therefore contains the following functions
  - enterSymbol(*name*, *type*) $\rightarrow$ adds the id *name* in the symbol table (current scope) with type *type*
  - retrieveSymbol(*name*) $\rightarrow$ returns the currently valid entry in the symbol table for *name* or a null pointer if no such entry exists
  - openScope() $\rightarrow$ opens a new scope so that any new symbols will be processed in the new scope
  - closeScope() $\rightarrow$ closes the current scope, so that all references revert to the outer scope
  - declaredLocally(*name*) $\rightarrow$ tests whether *name* is declared in the current scope

The Symbol Table Implementation

- The symbol table is an association list, capable of storing pairs key-data and retrieve stored data based on key values
  - Some additional complications are caused by the existence of scopes (to be addressed later)
- The usual suspects provide adequate implementations; the most efficient include
  - Balanced binary search trees $\rightarrow$ $O(\log n)$ access
    - Note that simple binary search trees will likely be inefficient since keys (variable names) are seldom random so the tree is likely to be unbalanced
  - Hash tables $\rightarrow$ particularly suited for implementing association lists, the most used data structure in practice

Type Checker Design

- Design process defining a type system:
  1. Identify the types that are available in the language
  2. Identify the language constructs that have types associated with them
  3. Identify the semantic rules for the language
- C++-like language example (declarations required = somewhat strongly typed)
  - Base types (int, double, bool, string) + compound types (arrays, classes)
    - Arrays can be made of any type (including other arrays)
    - ADTs can be constructed using classes (no need to handle them separately)
  - Type-related language constructs:
    - Constants: type given by the lexical analysis
    - Variables: all variables must have a declared type (base or compound)
    - Functions: precise type signature (arguments + return)
    - Expressions: each expression has a type based on the type of the composing constant, variable, return type of the function, or type of operands
    - Other constructs (if, while, assignment, etc.) also have associate types (since they have expressions inside)
  - Semantic rules govern what types are allowable in the various language constructs
    - Rules specific to individual constructs: operand to a unary minus must either be double or int, expression used in a loop test must be of bool type, etc.
    - General rules: all variables must be declared, all classes are global, etc.

Type Checking Implementation

- First step: record type information with each identifier
  - The lexical analyzer gives the name
  - The parser needs to connect that name with the type (based on declaration)
  - This information is stored in a symbol table
  - When building the node for a ⟨var⟩ construct (say, `int a;`) the parser can associate the type (`int`) with the variable (`a`)
  - A suitable entry in the symbol table can them be created
  - Typically the symbol table is stored outside the AST
  - The `class` or `struct` entry in a symbol table is a table in itself (recording all fields and their types)

$$
\begin{array}{rcl}
\langle \text{decl} \rangle & ::= & \langle \text{var} \rangle ;\ \langle \text{decl} \rangle \\
\langle \text{var} \rangle & ::= & \langle \text{type} \rangle\ \langle \text{identifier} \rangle \\
\langle \text{type} \rangle & ::= & \text{int} \\
& | & \text{bool} \\
& | & \text{double} \\
& | & \text{string} \\
& | & \langle \text{identifier} \rangle \\
& | & \langle \text{type} \rangle [\,]
\end{array}
$$

Type Checking Implementation

- First step: record type information with each identifier
  - The lexical analyzer gives the name
  - The parser needs to connect that name with the type (based on declaration)
  - This information is stored in a symbol table
  - When building the node for a ⟨var⟩ construct (say, `int a;`) the parser can associate the type (`int`) with the variable (`a`)
  - A suitable entry in the symbol table can them be created
  - Typically the symbol table is stored outside the AST
  - The `class` or `struct` entry in a symbol table is a table in itself (recording all fields and their types)

$$
\begin{array}{rcl}
\langle \text{decl} \rangle & ::= & \langle \text{var} \rangle ;\ \langle \text{decl} \rangle \\
\langle \text{var} \rangle & ::= & \langle \text{type} \rangle\ \langle \text{identifier} \rangle \\
\langle \text{type} \rangle & ::= & \text{int} \\
& | & \text{bool} \\
& | & \text{double} \\
& | & \text{string} \\
& | & \langle \text{identifier} \rangle \\
& | & \langle \text{type} \rangle [\,]
\end{array}
$$

- Second step: verify language constructs for type consistency
  - Can be done while parsing (in such a case declarations must precede use)
  - Can also be done in a subsequent parse tree traversal (more flexible on the placement of declarations)

Type Checking Implementation (cont'd)

- Second step: verify language constructs for type consistency, continued
  1. Verification based on the rules of the grammar
     - While examining an $\langle \text{expr} \rangle + \langle \text{expr} \rangle$ node the types of the two $\langle \text{expr} \rangle$ must agree with each other and be suitable for addition
     - While examining a $\langle \text{id} \rangle = \langle \text{expr} \rangle$ the type of $\langle \text{expr} \rangle$ (determined recursively) must agree with the type of $\langle \text{id} \rangle$ (retrieved from the symbol table)
     - Etc.

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{const} \rangle \\
& | & \langle \text{id} \rangle \\
& | & \langle \text{expr} \rangle + \langle \text{expr} \rangle \\
& | & \langle \text{expr} \rangle / \langle \text{expr} \rangle \\
& \ldots & \\
\langle \text{stmt} \rangle & ::= & \langle \text{id} \rangle = \langle \text{expr} \rangle \\
& \ldots &
\end{array}
$$

  2. Verification based on the general type rules of the language
     Examples:
     - The index in an array selection must be of integer type
     - The two operands to logical `&&` must both have `bool` type; the result is `bool` type
     - The type of each actual argument in a function call must be compatible with the type of the respective formal argument

- Essentially the process consists of annotating all AST nodes with type information, making sure that all annotations are consistent

Type Checking Implementation (cont'd)

- The AST annotation process is accomplished using synthesis rules
  - Specifies how to compute the type of a node from on the types of its children
- Examples include:
  - Various rules as specified in the language definition, e.g.
    if f has type $s \to t$ **and** x has type s **then** $f(x)$ has type t

Type Checking Implementation (cont'd)

- The AST annotation process is accomplished using synthesis rules
  - Specifies how to compute the type of a node from on the types of its children
- Examples include:
  - Various rules as specified in the language definition, e.g.
    if f has type $s \to t$ **and** x has type s **then** $f(x)$ has type t
  - Rules for type inference (if applicable), e.g.
    if $f(x)$ is an expression **then** for some α and β, f has type $\alpha \to \beta$ **and** x has type α
    - Type inference is necessary in languages such as ML and Haskell which do type checking but do not require declarations

Type Checking Implementation (cont'd)

- The AST annotation process is accomplished using synthesis rules
  - Specifies how to compute the type of a node from on the types of its children
- Examples include:
  - Various rules as specified in the language definition, e.g.
    if f has type $s \to t$ **and** x has type s **then** $f(x)$ has type t
  - Rules for type inference (if applicable), e.g.
    if $f(x)$ is an expression **then** for some α and β, f has type $\alpha \to \beta$ **and** x has type α
    - Type inference is necessary in languages such as ML and Haskell which do type checking but do not require declarations
  - Rules for type conversions (if allowed in the language), e.g.
    if $E_1.type = integer$ **and** $E_2.type = integer$ **then** $(E_1 + E_2).type = integer$
    else if $E_1.type = float$ **and** $E_2.type = integer$ **then** $(E_1 + E_2).type = float$
    ...

Type Checking Implementation (cont'd)

- The AST annotation process is accomplished using synthesis rules
  - Specifies how to compute the type of a node from on the types of its children
- Examples include:
  - Various rules as specified in the language definition, e.g.
    if f has type $s \to t$ **and** x has type s **then** $f(x)$ has type t
  - Rules for type inference (if applicable), e.g.
    if $f(x)$ is an expression **then** for some α and β, f has type $\alpha \to \beta$ **and** x has type α
    - Type inference is necessary in languages such as ML and Haskell which do type checking but do not require declarations
  - Rules for type conversions (if allowed in the language), e.g.
    if $E_1.type = integer$ **and** $E_2.type = integer$ **then** $(E_1 + E_2).type = integer$
    else if $E_1.type = float$ **and** $E_2.type = integer$ **then** $(E_1 + E_2).type = float$
    ...
  - Rules for overloaded functions, e.g.
    if f can have the type $s_i \to t_i$ for $1 \leq i \leq n$ with $s_i \neq s_j$ for $i \neq j$ **and** x has type s_k **then** $f(x)$ has type t_k

Type Checking Implementation (cont'd)

- Better (more general) approach to type conversions:
  - Establish a type hierarchy or partial order, based on the data storable in the types
    - $t_1 \leq t_2$ iff t_2 can store all the data storable in t_1
  - Define the function $\text{MAX}(t_1, t_2)$ which returns the least upper bound of t_1 and t_2 in the partial order
  - Define the function $\text{WIDEN}(a, t_1, w)$ which converts if necessary expression a from type t_1 to type w
    - If conversion is necessary then a new AST node will be inserted
    - If no conversion is necessary then the AST is not changed

if $E_1.type = t_1$ **and** $E_2.type = t_2$ **then**
    $w \leftarrow \text{MAX}(t_1, t_2)$
    if w is undefined **then** *signal type error*
    else
        $\text{WIDEN}(E_1, t_1, w)$
        $\text{WIDEN}(E_1, t_1, w)$
        $(E_1 + E_2).type \leftarrow w$

Scope Checking

- Scope constrains the visibility of an identifier to some subsection of the program
  - Local variables are only visible in the block in this they are defined
  - Global variables are visible in the whole program
- A scope is a section of the program enclosed by basic program delimiters such as { } in C
  - Many languages allow nested scopes
  - The scope defined by the innermost current such a unit is called the current scope
  - The scopes defined by the current scope and any enclosing program units are open scopes
  - All other scopes are closed
- Scope checking: given a point in the program and an identifier, determine whether that identifier is accessible at that point
  - In essence, the program can only access identifiers that are in the currently open scopes
  - In addition, in the event of name clashed the innermost scope wins

```
int a;                  // (1)
void bubble(int a) {    // (2)
    int a;              // (3)
    a = 2;              // (3) wins!
}
```

Implementation of Scope Checking

- Scope checking is implemented at the symbol table level, with two approaches
  1. One symbol table per scope organized into a scope stack
     - When a new scope is opened, a new symbol table is created and pushed on the stack
     - When a scope is closed, the top table is popped
     - All declared identifiers are put in the top table
     - To find a name we start at the top table and continue our way down until found; if we do not find it, then the variable is not accessible
  2. Single symbol table
     - Each scope is assigned a number
     - Each entry in the symbol table contains the number of the enclosing scope
     - A name is searched in the table in decreasing scope number (higher number has priority) $\rightarrow$ need efficient data organization for the symbol table (hash table)
     - A name may appear in the table more than once as long as the scope numbers are different
     - When a new scope is created, the scope number is incremented
     - When a scope is closed, all entries with that scope number are deleted from the table and then the current scope number is decremented

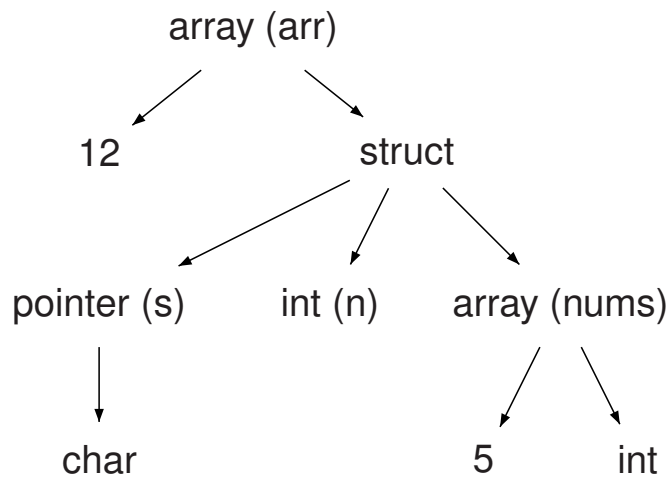
Implementation of Scope Checking (cont'd)

1. Stack of symbol tables
   - Disadvantages
     - Overhead in maintaining the stack structure (and creating symbol tables)
     - Global variables at the bottom of the stack $\rightarrow$ heavy penalty for accessing globals
   - Advantages
     - Once the symbol table is populated it remains unchanged throughout the compilation process $\rightarrow$ more robust code
2. Single symbol table
   - Disadvantages
     - Closing a scope can be an expensive operation
   - Advantages
     - Efficient access to all scopes (including global variables)

Symbol Tables: Advanced Features

- Compound types: types defined using other types, with arbitrary depth
  - Common storage technique: store compound types as a tree structure

```
struct {
    char *s;
    int n;
    int nums[5];
} arr [12];
```



  - Alternate technique: Each compound type entry is a symbol table by itself (containing the names and types of the members)
- Overloading: multiple ids with different type signatures
  - Possible storage techniques: have the id associated with a list of types (rather than a single type), or encode the type in the table key
- Type hierarchies: inheritance, interfaces, etc.

Equivalence of Compound Types

- Equivalence of compound types can be done recursively based on the tree structure

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
    if (tree1 == tree2) // if same type pointer, must be equivalent!
        return true;
    if (tree1->type != tree2->type) // check types first
        return false;
    switch (tree1->type) {
        case T_INT: case T_DOUBLE: ...  // same base type
            return true;
        case T_PTR:
            return AreEquivalent(tree1->child[0], tree2->child[0]);
        case T_ARRAY:
            return AreEquivalent(tree1->child[0], tree2->child[0]) &&
                   AreEquivalent(tree1->child[1], tree2->child[1]);
        ...
    }
}
```

Equivalence of Compound Types

- Equivalence of compound types can be done recursively based on the tree structure

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
    if (tree1 == tree2) // if same type pointer, must be equivalent!
        return true;
    if (tree1->type != tree2->type) // check types first
        return false;
    switch (tree1->type) {
        case T_INT: case T_DOUBLE: ...  // same base type
            return true;
        case T_PTR:
            return AreEquivalent(tree1->child[0], tree2->child[0]);
        case T_ARRAY:
            return AreEquivalent(tree1->child[0], tree2->child[0]) &&
                   AreEquivalent(tree1->child[1], tree2->child[1]);
        ...
    }
}
```

- Also needs some way to deal with circular types, such as marking the visited nodes so that we do not compare them ever again

Equivalence of Compound Types (cont'd)

- When are two custom types equivalent?
  - Named equivalence: when the two names are identical
    - Equivalence assessed by name only (just like base types)
  - Structural equivalence: when the types hold the same kind of data (possibly recursively)
    - Equivalence assessed by equivalence of the type trees (as above)
    - Structural equivalence is not always easy to do, especially on infinite (graph) types
- Named of structural equivalence is a feature of the language
  - Most (but not all) languages only support named equivalence
    - Modula-3 and Algol have structural equivalence.
    - C, Java, C++, and Ada have name equivalence.
    - Pascal leaves it undefined: up to the implementation

Type (Class) Definitions

- A class definition generated a new type just like for structures/records
  - However, this type also includes signatures for member functions
  - Using a symbol table for each class declaration more efficient than the tree implementation
- Need to maintain pointers to the parent classes (if any) and to the interfaces being implemented (if any)
- The pointers to the interfaces must be used to verify that all the interfaces are properly implemented
- The pointers to the parents will be used to resolve subsequent references to the members of the class