VarMutate: Dynamic Scoping for C using clang

Adarsh Patil

February 28, 2015

Abstract

Dynamic Scoping is the name visibility rule where the runtime state of the program stack determines what variable you are referring to. We use a technique in this work called VarMutate to implement dynamic scoping for C language using the clang compiler front end. Our implementation, VarMutate, dynamically creates a global typedef structure per variable name consisting of a union of all types a variable can take in the program and a tag to record the current meaning of the union. The algorithm then "mutates" this global typedef structures according the context to bind the variables referenced in the program. VarMutate is a single pass algorithm that traverses the AST created by the clang frontend. It does a source-source trasformation to the given input program and outputs a new C program that conforms to dynamic scoping rules. VarMutate is flexible enough to handle....

1 Introduction

Scoping of a name refers to the association of a name to an entity, such as a variable. Scope rules define the visibility rules for names in a programming language. For example, if you have references to a variable named a in different parts of the program do these refer to the same variable or to different ones? This is defined by the Scoping rules of a Programming language. Commonly there are 2 types of scoping rules used in languages. Static Scoping (sometimes referred to as Lexical Scoping) and Dynamic Scoping.

1.1 Static Scoping

In static scoping a block defines a new scope. Variables can be declared in that scope, and aren't visible from the outside. However, variables outside the scope – in enclosing scopes – are visible unless they are overridden. Most languages, including Algol, Ada, C, Pascal, Scheme, and Haskell, are statically scoped. In Algol, Pascal, Haskell, and Scheme (but not C or Ada) these scope rules also apply to the names of functions and procedures.

In other words, in static scoping the search for binding a name is first in the local function (the function which is running now), then you search in the

function (or scope) in which that function was defined, then you search in the function (scope) in which that function was defined, and so forth. "Lexical" here refers to text, in that you can find out what variable is being referred to by looking at the nesting of scopes in the program text.

We will demonstrate the scoping rules with the simple example of the below pseudo code:

```
begin
   int m,n
   m := 50;
   n := 60;
procedure hansel:
   begin
   print ("in procedure hansel -- ", m , n);
   end;
prodecure gretel:
   begin
   float m, n
   m := 3.145;
   n := 2.718;
   print ("in procedure gretel --", m , n);
   hansel;
   end;

print ("in main program");
gretel;
end;
```

Under static scoping rules the output of the above program will be

```
in main program
in procedure gretel -- 3.145 2.718
in proceedure hansel -- 50 100
```

This is because the the floating point varibles defined in gretel are available only within the scope of gretel. When gretel calls hansel, the variables in hansel are bound to the global declarations and initialization which are of type int.

1.2 Dynamic Scoping

Technically any scoping policy is dynamic if it is based on factors that can be known only when the program executes. Here we refer to dynamic scoping as follows: if a variable name's scope is a certain function, then its scope is the time-period during which the function is executing. While the function is running, the name exists, and is bound to its usages, but after the function returns, the name does not exist.

In other words, you search in the local function first, then you search in the function that called the local function, then you search in the function that

called that function, and so on, up the call stack. "Dynamic" refers to change, in the sense that the call stack can be different every time a given function is called, and so the function might hit different variables depending on where it is called from.

This means that if function f invokes a separately defined function g, then under lexical scoping, function g does not have access to f's local variables (assuming the text of g is not inside the text of f), while under dynamic scoping, function g does have access to f's local variables (since g is invoked during the invocation of f).

Now let us see the output of the same pseduo code under dynamic scoping

```
in main program
in procedure gretel -- 3.145 2.718
in prodecure hansel -- 3.145 2.718
```

The output for hansel is different this time due to the variables names being bound by looking at calling function i.e. values of variables m and n were got by looking down the call stack (activation records) rather than lexically.

2 Method

The two most popular methods for implementing DynamicScoping are Deep-Binding and ShallowBinding. Note that both of these strategies assume a last-in-first-out (LIFO) ordering to bindings for any one variable; in practice all bindings are so ordered.

2.1 Deep Binding

To find an identifier's value, the program can traverse the runtime stack, checking each activation record (each function's stack frame) for a value for the identifier. This is known as deep binding.

2.2 Shallow Binding

An alternate strategy, that is usually more efficient, is to maintain a central reference table of bindings which associates each name with its own stack of meanings for each identifier. This avoids a linear search during run-time to find a particular name, but care should be taken to properly maintain this table. The table is modified whenever the variable is bound or unbound, and a variable's value is simply that of the top binding on the stack. This is called shallow binding.

2.3 VarMutate Methodology

We uses a slight variant of the Shallow Binding method. The VarMutate method uses simple global variables to represent dynamic variables within a program.

The VarMutate method scans the AST of the program for all the variable names and tracks the various data types the variable can aquire over execution of the program. We then dynamically create a global typedef structure consiting of union of all the datatypes per variable. This is then used over the rest of the program to bind the names and values of variables. The current type of the variable is also tracked in this typedef struct with another tag member.

The local binding is performed by saving the original value in a backup variable. When that binding scope terminates, the original value is restored from this backup variable. In fact, Dynamic Scoping originated in this manner. Early implementations of Lisp used this obvious strategy for implementing local variables

As stated above, VarMutate is equivalent to the above shallow binding scheme, except that the central reference table is simply the global variable binding, in which the current meaning of the variable is the tracked in the global structure.

VarMutate is a single pass algorithm that parses the AST generated by the clang compiler. It maintains state information through various data structures (described later in the implementation section) for performing source-source transformation. VarMutate then adds the new typedef struct and renames the references of the variables to this new "mutating" struct and outputs a new C program that follows dynamic scoping rules.

3 Project Scope

There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called *local variables*,
2. Outside of all functions which is called *global variables*,
3. In the definition of function parameters which is called *formal parameters*.

For the purpose of this dynamic scoping implementation we limit ourselves to local and global variables only. Formal parameters are used to pass values from the caller to callee and in dynamic scoping such way of passing vslues is redudant as the variable names and values are themselves obtained at runtime by looking up the call sequence.

4 Implementation

We implement VarMutate in LLVM C Compiler (clang) . Input programs support the following the data types (a) int (b)float (c) struct (d) array

4.1 Clang codebase changes

We introduce a new data structure to the clang compiler to keep track of the undeclared but used variables. Since the clang generates the AST using static scoping rules it throws error messages and ignores all the lines where variables that are not accessible in that scope are referenced. This data structure is a vector with pair members that stores the DeclarationName and SourceLocation to all the undeclared but used occurances in the program.

It is added to the *include/clang/Sema/Sema.h* file.

```
835  /// Added by Adarsh
836  /// UndeclaredIdentifiers
837 std::vector<std::pair<DeclarationName, SourceLocation>>
    UndeclaredButUsed;
```

We also add a few lines to populate this datastructure over the parsing and semantic analysis of the input program. The below lines of code are added *clang/lib/Sema/SemaExpr.cpp* file.

```
1740 //Added by Adarsh
1741 if (diagnostic == diag::err_undeclared_var_use &&
1742   Name.getNameKind() == DeclarationName::Identifier) {
1743 UndeclaredButUsed.push_back(std::make_pair(Name, R.getNameLoc()));
1744 }
```

4.2 Classes

4.3 Features

1. **Multiple comma separated declarations in a single statement with initializations**, example:

   ```
   int a = 20,b;
   OR
   int a = (10*20+30); //complex expression
   ```

2. **Multiple (and repeated) undeclared variables in a single expression or statement**, example:

   ```
   a = b + c; (where a,b and c are undeclared in current scope)
   OR
   a = a * b;
   OR
   return (a+b);
   OR
   printf("%d",(a+b));
   ```

3. Can handle statements which have a **combination of DeclRef variables** (declared variable references) **and undeclaredButUsed variables** (variables undeclared but valid in context of dynamic scoping)

4. Can detect errors if variables that have never been declared anywhere in the program are used

5. adds code to detect run time error if referenced variable cannot be found in context of current function call, example:

```
int main()
{
   int var1 = 10;
   fun();
}
void game()
{
   fun();
}
int fun()
{
   // valid when called from main() invalid when called from game()
   var1 = var1 * 2;
}
```

6. Works with typedef definitions of variables.

7. Global variables (can be declared anywhere in global scope) and local variables with same name but different type in different scopes, examples:

```
int a;
//statements
void f()
{
    float a = 2.5;
    //statements
}
void f1()
{
      int a = 30;
      //statements
      if(<condition)
      {
         char a;
         //statements
      }
}
```

8. The code has a debug mode which prints complete log of the program for detailed analysis. Set {#define DEBUG 0} to turn off debug messages on stderr

9. The code has been refactored several times over multiple iterations to make it modular and designate functions to perform common operations.

4.4 Limitations

1. No support for arrays

2. if there is a return/break (unconditional branch) statement in the middle of the function the program will not be able to detect this and restore the global values of variables

3. Will not support single line if or for blocks without parathesis i.e. always use parantheis for if/switch/for/function blocks even if they are single line blocks

4. We place our dynamic stuct type all the way on top and if suppose a type struct is defined later we will get an error saying undefined struct.

5. including standard header files and printf will give error

5 About this code

5.1 How to Run

1. Edit the following parameters in the Makefile

   - LLVM_SRC_PATH := $$HOME/llvm/llvm-3.5.0.src
     This points to LLVM source which contains lib, include, tools etc.
   - LLVM_BUILD_PATH := $$HOME/llvm/build
     This points to the built directory of LLVM where make was run
   - LLVM_BIN_PATH := $(LLVM_BUILD_PATH)/Release+Asserts/bin/
     This points to the location of the binary of clang within the build directory

2. Once you make the modifications you can run *make* to build the code base

3. Running *make test* runs 6 test cases from the input directory and compares the output using python to test the built code

4. NOTE: Running make will add code to the clang code base and runs make on clang for the first run.

5.2 Important Notes

- This code runs only with LLVM and Clang 3.5 stable release WHY? - Clang and libtooling is rapidly evolving and the API calls change often. Trying to make it run for each version is like shooting a moving target!
- As state above running make will add code to the clang code base and runs make on clang for the first run.
  It then writes a hidden file .clangdone to the PWD for tracking purposes. Donot delete this file, doing so may lead to unstable code base.

6 Conclusion

Some languages like Emacs LiSP implement pure dynamic scoping rules. Quoting Peter Seibels popular book, Practical Common Lisp, in Seibels own words:

```
Dynamic bindings make global variables much more manageable, but
it's important to notice they still allow action at a distance.
Binding a global variable has two at a distance effects--it can
change the behavior of downstream code, and it also opens the
possibility that downstream code will assign a new value to a
binding established higher up on the stack. You should use  dynamic
variables only when you need to take advantage of one or both of
these characteristics.
```

Dynamic scoping does have some advantages:

- Certain language features are easier to implement.
- It becomes possible to extend almost any piece of code by overriding the values of variables that are used internally by that piece.

These advantages however come at a price:

- Since it is impossible to determine statically what variables are accessible at a particular point in a program, the compiler cannot determine where to find the correct value of a variable, necessitating a more expensive variable lookup mechanism. This lookup mechanism must be either space (stack, global variable) or time (activation record on stack lookup) intensive.
- Implicit extensibility makes it very difficult to keep code modular: the true interface of any block of code becomes the entire set of variables used by that block.

In this work we implemented dynamic scoping for C language using the described VarMutate methodlogy. This implementation adds code to the source program in the form of global structures and modifies declaration refereces to refer to this structure. Thus the overall complexity of the program increases manyfold and several additional lines of code are added to program. VarMutate

increases the storage needed as we also keep a tag to interpret the type of the name begin refered albeit only by a few bytes per variable. VarMutate also increases the time complexity of execution of the program.

7 References

1. The complete code of implementation is available at `https://github.com/adarshpatil/dynamicScopingForC`