

Forum:[Hack Forums](#)

Thread: [\[Advanced\] \[C\] Writing a Metamorphic Engine / Real USG; Part 3](#)

movsx - September 10, 2017, 6:02 pm

It's been a while since my last tutorial, I've been very busy with my job, as a compiler engineer, and personal life. Anyway, here's the third part. I'm trying to keep the theory very dry and make the topics as practical as possible, so the reader can understand easily.

Topics

Symbol Tables

Semantic Analysis

Syntax-directed translation and Intermediate Representation as 3-Address Code

Intermediate Representation Generation

Symbol Tables

A symbol table can be thought of a hash table. There will be one symbol table per scope (including the global scope, containing all global variables and global functions) and it will map an identifier in that scope to specific information about that identifier such as its attributes (constness, static-ness, etc), its value, and other relevant information (for the purpose of this topic, nothing else is relevant). Symbol tables will be used in later stages of the engine. We will use a cactus stack of hash tables to implement our symbol table. A cactus stack is required so we can easily change the focus scope symbol table as we're working on our engine in later stages.

Semantic Analysis

In our prior stages, we generated a parse and used syntax-directed translation (albeit it wasn't mentioned; more on this later) to generate an AST. We also were able to determine if our language were syntactically correct. What this means, is we now have a language that has correct syntax, but it is possible it has incorrect semantics.

It's always helpful to draw on parallels from prior knowledge here. Let's look into the English language to help explain this.

The sentence "The stapler is gay." is syntactically correct, but not semantically correct. A stapler cannot be gay.

Let's try to draw a parallel into the C programming language here:

I.e., here's some examples of a syntactically correct series of statements, but not semantically correct.

```
Code const int i = 0;
```

```
i = 5;
```

```
// invalid C code
```

```
Code const int i = 5;
```

```
const int i = 6;
```

```
// invalid C code
```

```
Code const int i = 5 + j; // j is not defined yet
```

```
// invalid C code
```

```
Code const float f = 5 / 6;
// semantically incorrect, but still valid C code
Code const char* const s = "hello";
int j = s + 1;
// semantically incorrect, but still valid C code
Code const char* const s = "hello";
const char* const p = "world";
const int j = s * p;
// semantically incorrect
```

In general, we are concerned about type checking, correct environment usage, and language-specific definitions. But this has little to do with metamorphic engines, and more to do with compiler theory. Therefore I won't discuss this section further. Implementing code for semantic analysis is typically done via the visitor design pattern or in procedural languages via postorder tree traversals on the AST. It is also very clear we'd be using our cactus stack of symbol tables while doing this stage, changing the focused symbol table as we visit our AST and enter or leave specific scopes.

Syntax-directed translation and Intermediate Representation as 3-Address Code

We used this in our last tutorial (although it wasn't mentioned) to transform our parse tree into an AST. In this stage, we will use syntax-directed translation (SDT) to transform our AST into an intermediate representation. Of course, there's a ton of theory surrounding SDT, but it will be ignored for the sake of practicality.

Think of SDT as some code which takes some graph data structure as input and generates another data structure as output by attaching an action rule to each context-free grammar derivation. In our last tutorial, we attached rules to our context-free grammar's procedural implementation functions to convert our parse tree into an abstract-syntax tree; although those rules were omitted and only the resulting code was shown, for the sake of clarity and practicality.

Our goal here is to transform C code into BOTH (1) a linear representation AND (2) a linear directed-acyclic graph representation. (2) will mostly be used for optimizations and permutation generation later on in our engine, as well as instruction scheduling. Why do we care about instruction scheduling? Because we want to preserve dependencies in our code. We would use a typical modified topological sorting algorithm on our directed-acyclic graph 3-address code to ensure dependency relationships were constrained. We could also permute the ordering of instructions to generate unique code, so as long as these dependency relationships were constrained (to be discussed in later tutorials).

Very simple. Say we have the x86 struct... Code struct {int* ptr; int a;} demo; and edx's value is the address of demo on the stack.

(1) Set the pointer member of the structure to point to its second member

```
Code lea eax, [edx+4]
mov [edx], eax
is very different from ...
```

(2) Fill the pointer member of the structure with garbage data

```
Code mov [edx], eax
lea eax, [edx+4]
More on 3-address code:
```

All 3-address code will follow this general structure and we will use a unique destination for each new statement.

Code (1) DST = SRC1 BINOP SRC2

(2) DST = UNOP SRC1
(3) IF CONDITION GOTO LABEL
(4) IF NOT CONDITION GOTO LABEL
(5) GOTO LABEL
(6) LABEL:

We can parallel these to the following x86 instructions:

Code (1)
MOV DST, SRC1
BINOP DST, SRC2

(2)
MOV DST, SRC1
UNOP DST

(3) and (4)
;EVAL CONDITION
JCC LABEL

(5)
JMP LABEL

Essentially you can think of it as a high-level assembly code, with an infinite number of registers. Later on, we solve the problem of mapping infinitely many registers into a finite number of registers through the process of register allocation

Here is an example of both linear and DAG 3-address code in play:

The purpose of using an intermediate representation will become clear, if not already, in later tutorials. It is necessary and the way of compiler writing. You take a big problem, and you simplify it. Look at what we've been doing so far. Can you imagine how hard it would be to write a metamorphic engine if you only had to work with a giant character array? Damn near impossible!

Intermediate Representation Generation

Here we establish rules that we will use to convert C code into 3 address code. Note that it's not complete, and that we'd use a SDT to do the actual conversions for us. I show conversion by example here and trust that the reader isn't retarded and can figure out how to code this themselves. (I already mentioned earlier, post order traversal... and a bunch of if statements plus symbol table lookups!)

Code int a = 5;

int b = 6;

int c = a + b;

if (c > 2*b) {

 c = 1;

}

else {

 c = 0;

}

Code t1 = 5

t2 = 6

t3 = t1 + t2

t4 = 2 * t2

t5 = t3 > t4

if not t5 goto l1:

```
t3 = 1  
goto l2  
l1:  
t3 = 0
```

l2: Note we don't apply any optimizations above... i.e. $t4 = t2 \ll 1$ is an optimization, and there's a ton more we can optimize. But we don't do that here, nor is optimization the purpose of a metamorphic engine.

It should also be clear to an astute reader that we could easily convert the given intermediate representation code into assembly with a little more work!

If this tutorial is well-liked, I will post future tutorials.

In future tutorials, I will go back to intermediate representation generation to discuss the implementation of junk code insertion, instruction scheduling, optimization, and other techniques involving intermediate representations.

Then I will post a tutorial on final code optimization and register allocation (graph coloring). Finally, a tutorial will be posted on instruction selection as graph covering.

SIM-2263652 - *September 13, 2017, 1:38 am*

Good tutorial, nice work mate. Liking the visual representations of the lin/dag code.

thesnipair.24 - *August 3, 2018, 2:20 am*

Great high quality tutorial, really looking forward for the next part. cheers mate :)