Control Flow Graphs

Having seen how ASTs are assembled, we can start turning to matters of control flow. At this point, we are going to turn the AST into something that we can execute – or at least something whose execution we can simulate. I apologize in advance for the length of this note. The end result is quite simple, but you will need to see how it is achieved in order for it to make any sense.

1. From AST to Pseudo-Code

Before we can generate a CFG, we first need to generate what you might think of as “pseudo-instructions.” We have started to talk about this in class, but the story I am going to tell here is a little different from the one in class. In the story here (which is the one we should use), each pseudo-instruction has the effect of completing a single AST. Our immediate task is to perform code generation for each of the procedures that we have. Each procedure definition symbol is going to end up with a code pointer, and the code pointer is going to point to the first pseudo-instruction in the procedure.

1.1. Expression Generation

One of the simplest ways of doing code generation is to emit directly from the AST tree. If you prefer, you can imagine this as running an interpreter using the AST as an intermediate form. If you want optimization, this isn't a clever thing to do, but it works beautifully for our purposes. Consider the following example:

\[
\text{a} = \text{b} + \text{f}(\text{x}, \text{g(y)});
\]

We ought to get something like:

\[
\{ \text{set } \{ \text{ident “a” } \} \\
\{ \text{plus } \{ \text{ident “b” } \} \\
\{ \text{funcall “f” } \\
\{ \text{ident “x” } \} \\
\{ \text{funcall “g” } \\
\{ \text{ident “y” } \} \} \} \}
\]

One way to execute this is to simply do an in-order traversal. To execute a given AST node, first execute its children left to right and then execute the node itself. Ultimately, we will find some “leaf” AST-nodes that are either identifier names or literals. The “execution” of these can be handled directly.

As a preliminary step, however, it is useful to assign unique identifiers to each and every AST node:
Having done so, we can write out the order in which these AST nodes will be executed according to the ordering rule above:

\[ 2 \ 4 \ 6 \ 8 \ 7 \ 5 \ 3 \ 1 \]

Here is the tricky part: *for this class we don't care about the execution*. In a normal compilation, we would worry here about generating code that would perform the computations. For purposes of static analysis, we are only concerned about knowing the order in which the ASTs *would* get executed if we were actually executing the program. So we can generate pseudo-instructions directly from the AST that might look like:

\[
\begin{align*}
( &1 \text{ exec ast:2 pi:2 }) \\
( &2 \text{ exec ast:4 pi:3 }) \\
( &3 \text{ exec ast:6 pi:4 }) \\
( &4 \text{ exec ast:8 pi:5 }) \\
( &5 \text{ exec ast:7 pi:6 }) \\
( &6 \text{ exec ast:5 pi:7 }) \\
( &7 \text{ exec ast:3 pi:8 }) \\
( &8 \text{ exec ast:1 pi:next })
\end{align*}
\]

Where each pseudo-instruction takes the form:

\[
( \text{unique-id pseudo-instr-type completed-ast-id next-pi-id})
\]

Boring, huh?

### 1.2. If Statements

So now let's look at a slightly more interesting example involving conditionals. The code is:

```java
if (a > 5)
  x = x + 5;
else
  x = x - 1;
```

Assuming here that the if-then-else AST takes the form:

```java
{ if predicate-ast then-ast else-ast }
```

Then the AST for this looks like:
The question is: how do we generate code for this? The answer is: first emit code for the predicate, then for the then block, then for the else block, but the predicate code generation is going to end with a new pseudo-instruction called “cond”:

```plaintext
// code for predicate:
( 1 exec 3 pi:2 )
( 2 exec 4 pi:3 )
( 3 exec 2 pi:4 )
( 4 exec 1 pi:5 )    // see note below
( 5 cond pi:6 pi:11 )

// code for then-block:
( 6 exec ast:6 pi:7 )
( 7 exec ast:8 pi:8 )
( 8 exec ast:8 pi:9 )
( 9 exec ast:7 pi:10 )
( 10 exec ast:5 pi:16 ) // next instruction is after the if

// code for else-block:
( 11 exec ast:11 pi:12 )
( 12 exec ast:13 pi:13 )
( 13 exec ast:14 pi:14 )
( 14 exec ast:12 pi:15 )
( 15 exec ast:10 pi:16 )

// code for next statement:
( 16 some-instr ... pi:next )
```

Note that we are not trying to do anything fancy at this point. No goto elimination or any such thing. Note also that from the code above you should be able to easily work out how to encode expressions such as:

```
a ? (b += 5) : (c += 7)
```

You should also be able to work out how to emit “for” loops and similar control structures.

Now notice that there is another way to think about the code above. Each “exec” instruction can be thought of as “emitting” the identity of the AST whose completion it represents. That is, execution of these pseudo programs produces strings whose tokens are uniquely identified ASTs in the source code. We will come back to this in a week or two.

One part of the code generated above is tricky. Look again at pseudo instruction 4, which claims that we have executed the “if” statement AST. In a strict bottom-up construction this would not be true. Conceptually, we haven’t finished the “if” statement until we have completed either the then block or the else block. From a modeling perspective, however, we
probably want to think of the sequence of execution as “if then else”. That is, the if conditional is executed before its corresponding then or else block. Arguments can be made either way, and you begin to see here why matching ASTs isn't always enough.

1.3. Call and Return

We will need two special instruction types: call and return. A “call” instruction is really just an exec instruction whose corresponding AST is a function call AST. We want a distinct instruction mainly because we are going to want to handle this specially. Similarly, a “return” instruction is an exec instruction whose AST is a procedure return. Eventually, it may be convenient to introduce distinct instructions for “if-start” and “if-complete” as well, but let's leave that for later.

Once we have the instructions generate, we will make a pass over them looking for exec instructions whose corresponding AST is a function call AST. We will change these to call instructions. Similarly, we will convert exec instructions whose AST is some form of procedure return into return instructions. If you like, you can do this while you are creating the pseudo instruction sequence in the first place (probably might as well).

2. Simulating the Program

Now that we have pseudo-code, we can simply start at “main” and execute the program by following the pseudo instructions. As we execute, we will emit AST identifiers, and these will get fed into the FSA state machine to look for an error. Actually, we can't really do this. There are three problems to watch out for: loops, conditionals, and procedure calls. But we can get close enough!

2.1. Executing Loops

The really good news about our simulated execution is that we aren't interested in the values of the program at all. In fact, static analysis is never concerned about values, because these cannot be known statically.\(^1\) What we are concerned about is things that change the FSA state.

This means that if we execute a given instruction in a given state, and later we end up coming back to that instruction, we may not need to run the instruction again. If we are re-executing in the same FSA state that we were in before, then our simulated execution will proceed in the same way as it did before, and we don't need to do it again. We only need to re-execute the instruction if we are now in a different state.

This means that we can think of the simulated execution as evaluating a work list of pairs

---

\(^1\) In really advanced tools, we may perform some amount of symbolic execution to try to resolve suspicious traces. For example if (x) lock() followed shortly by if (x) unlock(). Even in these cases we won't know what the values are, but we may be able to show that whatever they were they could not have changed between the lock() and the unlock(), and therefore the lock() and the unlock() are a properly matched pair.
(FSA-state, initial-instruction)
where the goal of the execution is to determine a pair of the form
(FSA-state, next-instruction)
that identifies the end of the trace that started at initial-instruction. Each (FSA-state, next-instruction) pair will then get stuck on the work list for further processing. As we initiate the simulation of each pair on the work list, we remember that we have already done this one. If we later need to re-simulate the same trace in the same state, we'll just take the old answer. In the next section, I'll talk about how to figure out where the end of each trace is.

2.2. Executing Conditionals
The problem with conditionals is that we aren't really executing the program, so we don't really know which way the conditional is going to go. What we do here is make a conservative assumption: we assume that there exists possible executions in which the conditional will go to the “then” clause, and other possible executions in which the conditional will go to the “else” clause. If this isn't possible in the real program, then there probably shouldn't be a conditional at all, but we are merely trying to validate that both sides of the conditional satisfy whatever property we are trying to test.

So when we reach a conditional, there are two possible future “traces” of the execution: one proceeding left and the other proceeding right. So here is what we do: instead of trying to build one huge long trace through the entire program, we are going to build a lot of short traces, each of which ends at a conditional. Later, if we find an error, we will thread these traces together to determine what went wrong.

2.3. Executing Procedure Calls
It turns out that we also need to start new traces at the entry to each procedure call. This may seem counter-intuitive. In the case of conditionals, the reason to “split” a trace was because we did not know if it would go left or right. When we make a virtual procedure call, we similarly have a sort of a “split” that can happen in the control flow depending on the object type. But when we execute a non-virtual procedure call the execution proceeds in only one direction. Why do we need to generate multiple traces?

The answer is that a normal instruction can only change the program from one input state to a single output state, but a procedure call may have multiple paths, and these paths may exit the procedure in distinct output states. Therefore, a call to a procedure that is made in a given input state may result in multiple output states. Conceptually, what we need to do here is start a new trace at the return instruction in every possible state that the procedure call might generate.

Here, however, we run into another puzzle: we do not know what these states will be until we simulate the execution of the function. Yuck! Functions are going to need some sort of special handling. What we are going to do is remember all of the places where each function was called
and add these traces to our work list every time the function returns. What we will do is simulate something very much like a stack. The result is that our simulator is acting as a push-down automata.

What we do at this point is that we associate a different kind of work list with every function that gets called. I will call this the function return site list. The return site work list is simply a list of all instructions to which this function is known to return. In addition, we will keep a set of pairs for each function of the form (from-state, to-state). This is the function transition list. Whenever we see a function call, we will be looking at an instruction sequence of the form:

( 7 exec ast:8 pi:8 )
( 8 CALL ast:8 pi:9 )
( 9 exec ast:7 pi:10 )  // next instruction after the call
( 502 exec ast:872 pi:503 )  // first instruction of called procedure
...  
( 587 RETURN ast:997 )  // note that RETURN does not have a next-pi

In the description below, I will assume that we are calling some function f().

When we encounter a CALL instruction in some FSA state s, we will proceed as follows:

1. Add the instruction after the call to the function return site list of the destination function. This is the instruction that the function will be returning to. In this case, we are going to add pi:9 to the return site list of the function f().

2. Add the pair ( pi:502, s ) to the work list, because we need to execute the first instruction of the function in the current state. It is possible that we have already done so, but we'll take care of that later.

3. Examine the function transition list looking for transitions where the from state is s. For every corresponding result state, add the pair ( next-insn, result-state ) to the work list.