15-411: Compiler Design — Fall 2018

Recitation 4: Lab 2, From AST to Assembly — 28 September

Static Semantics

For a C0 program to be valid, all variables must be declared and initialized before use. A compiler should confirm this property of a user program. To formally describe what the compiler needs to derive, we will give inference rules for some important judgments. Here are English-language intuitions for what the judgments mean:

- $\mathsf{use}(e, x)$: the variable x *might* be used when evaluating expression e.
- $\mathsf{def}(s, x)$: the variable x *must* be initialized when executing statement s.
- $\mathsf{live}(s, x)$: x *might* be used before initialization when executing s.
- $\mathsf{init}(s)$: all variables declared in s *must* be initialized before use in s.

The compiler attempts to derive $\mathsf{init}(s)$ for the user program; and, if this derivation fails, the compiler reports that the program is invalid. Some inference rules for $\mathsf{init}(s)$ follow:

$$\frac{}{\mathsf{init}(\mathsf{nop})} \qquad \frac{\mathsf{init}(s_1) \quad \mathsf{init}(s_2)}{\mathsf{init}(\mathsf{seq}(s_1, s_2))} \qquad \frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))}$$

It is instructive to read these rules going from the conclusion to the premises ("bottom-up"). For example, we might say: "$\mathsf{init}(s_1, s_2)$ holds if we can show that $\mathsf{init}(s_1)$ and $\mathsf{init}(s_2)$ hold."

Checkpoint 0

What premises are necessary to derive $\mathsf{init}(\mathsf{assign}(x, e))$? Write the inference rule.

Likewise, we can give some rules for $\mathsf{live}(s, x)$:

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{ret}(e), x)} \qquad \frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)} \text{ (Note that we do not constrain } x \neq y.)$$

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

Again, reading these rules from the bottom up: "x could be used before initialization in $\mathsf{while}(e, s)$ if x is used in e. This would also hold if x could be used before initialization in s."

Checkpoint 1

Write the inference rules for $\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)$.

Checkpoint 2

Using the inference rules as given, try to derive $\mathsf{init}(s)$ for the following two programs:

- $\mathsf{decl}(x, \texttt{int}, \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{ret}(x)))$
- $$\begin{array}{l}\mathsf{decl}(x, \texttt{int}, \\ \mathsf{decl}(y, \texttt{int}, \\ \quad \mathsf{seq}(\\ \quad\quad \mathsf{if}(\texttt{true}, \mathsf{assign}(x, 1), \mathsf{assign}(x, 2)), \\ \quad\quad \mathsf{ret}(x + y))))\end{array}$$

Traversing the same AST multiple times to derive multiple facts can be inefficient. It would be nice if we could derive all facts at once as we traverse the AST.

What we want is a database of facts that we can update as we traverse the AST. The initialization-checking function would now take in the current database and return the updated database (throwing an exception if the statement is not legal given the current database). The ML type might look like:

```
val initialized : database * statement -> database
```

When deciding on the `database` type, we should pay careful heed to this rule from above, which checks two separate properties of a statement:

$$\frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))}$$

That is, our database type should combine the init and ¬live judgments in such a way that we can check both properties at once as we traverse the AST. Combining our informal descriptions from before, that property might sound like: "all variables used in this statement have either already been initialized or will be initialized before use."

Checkpoint 3

What type should we define `database` as to be able to check this property as we traverse the AST?

Checkpoint 4

Rewrite the inference rule given on this page as a case of the `initialized` function whose type is given above.

Checkpoint 5

So far, we have only talked about whether a variable is initialized before use, not about whether a variable is declared before initialization. Will checking for correct declarations require a separate pass? Is it possible to update the `database` type to carry the necessary information for `initialized` to additionally check for declarations?

In the next section, we will consider the following L2 program:

```
1 // compute the log base 10 of a number
2 int main() {
3   int input = 500; // the "input"
4   int output;
5   if (input <= 1) return 0;
6   for(output = 1; input > 0; output++) {
7     input /= 10;
8   }
9   return output;
10 }
```

Parsing vs. Elaboration

A number of design decisions in Lab 2 center around the roles and responsibilities of the parsing and elaboration steps. In general, we recommend that you keep your parser as simple as possible; this means that the AST generated by your parser should reflect the source-code **syntactic** structure of L2 as closely as possible. Once you've parsed the program and you have a parse tree, you can then perform an elaboration pass to generate an AST that reflects the **semantic** structure of L2.

For example, the Lab 2 handout recommends that you use a case of the form declare(x, τ, s) to represent the scope of a variable x, but you don't need to use this exact form as part of your parser. Instead, you can parse declarations as regular statements, and then explicitly represent their scope during elaboration.

Of course, you're welcome to design your compiler to divide the responsibilities differently (or even not have an elaboration pass at all). However, the structure we're recommending has worked well for students in the past. And, as with most things in this course, you will likely need to revisit most of your choices for later labs.

Checkpoint 6

Given the following parse tree representing the program above, perform an elaboration that captures variable scopes, transforms the `for` loop into a `while` loop, and replaces increment and arithmetic assignment operators with more primitive constructs.

```
Seq(Init(input, int, 500),
  Seq(Declare(output, int),
    Seq(If(Compare(input, 1, LEQ),
        Return(0),
        Nop()
      ),
      Seq(For(
          Assign(output, 1),
          Compare(input, 0, GT),
          Incr(output),
          Block(
            AssignOp(input, 10, DIV)
          )
        ),
        Return(output)
      )
    )
  )
)
```

Checkpoint 7

Fill in the missing instructions in our IR tree. Then, identify all of the basic blocks.

```
_c0_main:
  input <- 500
  if _______________ then goto _____ else goto _____
.L0:
  return(0)
  goto L2
.L1
  goto L2
.L2
  output <- 1
.L3
  if _______________ then goto _____ else goto _____
.L4
  input <- ___________________
  output <- ___________________
  goto _____
.L5
  return(output)
```