Lecture Notes on
Static Single Assignment Form

15-411: Compiler Design
Frank Pfenning, Rob Simmons

Lecture 10
October 1, 2015

1 Introduction

In abstract machine code of the kind we have discussed so far, a variable of a given name can refer to different values even in straight-line code. For example, in a code fragment such as

$$\begin{array}{lcl} 1 & : & i \leftarrow 0 \\ \ldots & & \\ k & : & \mathsf{if}\ (i < 0)\ \mathsf{then\ error\ else\ continue} \end{array}$$

we can apply *constant propagation* of 0 to the condition (turning into a goto continue) only if we know that the definition of i in line 1 is the only one that reaches line k. It is possible that i is redefined either in the region from 1 to k, or somewhere in the rest of the program followed by a backwards jump. It was the purpose of the *reaching definitions* analysis in to determine whether this is the case. If lines 1-k are part of a single basic block, then our task is simpler: we only need to check whether there is an intervening definition of i within the basic block.

An alternative is to relabel variables in the code so that each variable is defined only once in the program text. If the program has this form, called *static single assignment* (SSA), then we can perform constant propagation immediately in the example above without further checks. There are other program analyses and optimizations for which it is convenient to have this property, so it has become a de facto standard intermediate form in many compilers and compiler tools such as LLVM.

In this lecture we develop SSA, first for straight-line code and then for code containing loops and conditionals. Our approach to SSA is not entirely standard, although the results are the same on control flow graphs that can arise from source programs in the language we compile.

2 Basic Blocks

As before, a *basic block* is a sequence of instructions with one entry point and one exit point. In particular, from nowhere in the program do we jump into the middle of the basic block, nor do we exit the block from the middle. In our language, the last instruction in a basic block should therefore be a return, goto, or if. On the inside of a basic block we have what is called *straight-line code*, namely, a sequence of moves or binary operations.

It is easy to put basic blocks into SSA form. For each variable, we keep a *generation counter* to track which definition of a variable is currently in effect. We initialize this to 0 for any variable live at the beginning of a block. Then we traverse the block forward, replacing every use of a variable with its current generation. When we see a redefinition of variable we increment its generation and proceed.

As an example, we consider the following C0 program on the left and its translation into a single basic block on the right.

```
int dist(int x, int y) {
  x = x * x;
  y = y * y;
  return isqrt(x+y);
}
```

```
dist(x,y):
  x <- x * x
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

Here `isqrt` is an integer square root function previously defined. We have assumed a new form of instruction

$$d \leftarrow f(s_1, \ldots, s_n)$$

where each of the sources s_i is a constant or variable, and the destination d is another variable. We have also marked the beginning of the function with a parameterized label that tracks the variables that may be live in the body of the function.

The parameters x and y start at generation 0. They are *defined implicitly* because they obtain a value from the arguments to the call of dist.

```
dist(x0,y0):
  ------------- x/0, y/0
  x <- x * x
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

We mark where we are in the traversal with a line, and indicate there the current generation of each variable. The next line uses x, which becomes x_0, but is also defines x, which therefore becomes the next generation of x, namely x_1.

```
dist(x0,y0):
  x1 <- x0 * x0
  ------------- x/1, y/0
  y <- y * y
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

The next line is processed the same way.

```
dist(x0,y0):
  x1 <- x0 * x0
  y1 <- y0 * y0
  ------------- x/1, y/1
  t0 <- x + y
  t1 <- isqrt(t0)
  return t1
```

At the following line, t_0 is a new temp. The way we create instructions, temps are defined only once. We therefore do not have to create a new generation for them. If we did, it would of course not change the outcome of the conversion. Skipping ahead now, we finally obtain

```
dist(x0,y0):
  x1 <- x0 * x0
  y1 <- y0 * y0
  t0 <- x1 + y1
  t1 <- isqrt(t0)
  return t1
```

We see that, indeed, each variable is defined (assigned) only once, where the parameters x_0 and y_0 are implicitly defined when the function is called and the others explicitly in the body of the function. It is easy to see that the original program and its SSA form will behave identically.

3 Loops

To appreciate the difficulty and solution of how to handle more complex programs, we consider the example of the exponential function, where $\mathsf{pow}(b, e) = b^e$ for $e \geq 0$.

```
int pow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    // r*b^e remains invariant
    {
      r = r * b;
      e = e - 1;
    }
  return r;
}
```

We translate this to the following abstract machine code, which is comprised of basic blocks:

```
pow(b,e):
  r <- 1
loop:
  if (e > 0) then done else body
body:
  r <- r * b
  e <- e - 1
  goto loop
done:
  return r
```

There are two ways to reach the label loop: when we first enter the loop, or from the end of the loop body. This means the variable e in the conditional branch really could refer to either the procedure argument, or the value of e after the decrement operation in the loop body. Therefore, our straightforward idea for SSA conversion of straight line code no longer works.

The key idea is to parameterized labels (the jump targets) with the variables that are live in the block that follows. The variant of liveness necessary here can be calculated with respect to the block-structured AST – it is not necessary to use the more potentially expensive liveness analysis based on dataflow. One can also safely, but redundantly, just use all variables, or all variables that are declared and

defined at that point in the program. We use this approach in the following example. Labels l occurring as targets in goto l or if $(-)$ then l else l' are then given matching arguments.

```
pow(b,e):
  r <- 1
  goto loop(b,e)

loop(b,e,r):
  if (e > 0)
    then body(b,e,r)
    else done(b,e,r)

body(b,e,r):
  r <- r * b
  e <- e - 1
  goto loop(b,e,r)

done(b,e,r):
  return r
```

Next, we convert each block into SSA form with the previous algorithm, but using a global generation counter throughout. An occurrence in a label in a jump goto $l(\ldots, x, \ldots)$ is seen as a *use* of x, while an occurrence of a variable in in a jump target $l(\ldots, x, \ldots)$ is seen as a *definition* of x. Applying this to the first block we obtain

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)
-------------------- b/0, e/0, r/0
loop(b,e,r):
  if (e > 0)
    then body(b,e,r)
    else done(b,e,r)

body(b,e,r):
  r <- r * b
  e <- e - 1
  goto loop(b,e,r)

done(b,e,r):
  return r
```

Since we encounter a new definition of b, e, and r we advance all three generations and proceed with the next two blocks.

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 > 0)
    then body(b1,e1,r1)
    else done(b1,e1,r1)

body(b2,e2,r2):
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop(b2,e3,r3)
-------------------- b/2, e/3, r/3
done(b,e,r):
  return r
```

Continuing through both the last block, we obtain:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 > 0)
    then body(b1,e1,r1)
    else done(b1,e1,r1)

body(b2,e2,r2):
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop(b2,e3,r3)

done(b3,e4,r4):
  return r4
```

First, we verify that this code does indeed have the SSA property: each variable is assigned at most once, even counting implicit definitions at the parameterized labels pow(b_0, e_0), loop(b_1, e_1, r_1), body(b_2, e_2, r_2), and done(b_3, e_4, r_4).

The operational reading of this program should be evident. For example, if we reach goto loop(b_2, e_3, r_3) we pass the current values of b_2, e_3 and r_3 and move them

into variables b_1, e_1, and r_1. That fact that labeled jumps correspond to moving values from arguments to label parameters will be the essence of how to generate assembly code from the SSA intermediate form in Section 7.

4 SSA and Functional Programs

We can notice that at this point the program above can be easily interpreted as a *functional program* if we read assignments as bindings and labeled jumps as function calls. We show the functional program below on the right in ML-like form.

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)


loop(b1,e1,r1):
  if (e1 > 0)
    then body(b1,e1,r1)
    else done(b1,e1,r1)

body(b2,e2,r2):
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop(b2,e3,r3)


done(b3,e4,r4):
  return r4
```

```
fun pow (b0, e0) =
   let val r0 = 1
   in loop (b0, e0, r0)
   end

and loop (b1, e1, r1) =
   if e1 > 0
      then body (b1, e1, r1)
      else done (b1, e1, r1)

and body (b2, e2, r2) =
   let val r3 = r2 * b2
       val e3 = e2 - 1
   in loop (b2, e3, r3)
   end

and done (b3, e4, r4) =
   r4
```

There are several reasons this works in general. First, in SSA form each variable is defined only once, which means it can be modeled by a let binding in a functional language. Second, each goto is at the end of a block, which translates into a tail call in the functional language. Third, because all jumps become tail calls, a return instruction can be modeled simply be returning the corresponding value.

We conclude that translation into SSA form is just translating abstract machine code to a functional program! Because our language does not have first-class functions, the target of this translation also does not have higher-order functions. Interestingly, this observation also works in reverse: a (first-order) functional program with tail calls can be translated into abstract machine code where tail calls become jumps.

While this is clearly an interesting observation, it does not directly help our compiler construction effort (although it might if we were interested in compiling a functional language).

5 Optimization and Minimal SSA Form

At this point we have constructed clean and simple abstract machine code with parameterized labels. But are all the parameters really necessary? Let's reconsider:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 > 0)
    then body(b1,e1,r1)
    else done(b1,e1,r1)

body(b2,e2,r2):
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop(b2,e3,r3)

done(b3,e4,r4):
  return r4
```

There is no need to pass b_1, e_1, and r_1 to body and assign their values to b_2, e_2, and r_2 (respectively). Instead, we could remove these arguments and instead substitute b_1 for b_2, e_1 for e_2, and r_1 for r_2. The same goes for the arguments to done, though we could also conclude that b_3 and e_4 are unnecessary because those temps are never even live. This yields:

```
pow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):
  if (e1 > 0)
    then body()
    else done()

body():
  r3 <- r1 * b1
  e3 <- e1 - 1
  goto loop(b1,e3,r3)

done():
  return r1
```

We see this is still in SSA form. Next we can ask if all the arguments to loop are really necessary. We have two gotos and one definition:

```
goto loop(b0,e0,r0)
goto loop(b1,e3,r3)

loop(b1,e1,r1):
```

Let's consider the first argument. In the first call it is b_0 and in the second b_1. Since we have SSA form, we know that the b_1 will always hold the same value. In fact, the only call with a different value is with b_0, so b_1 will in fact always have the value b_0. This means the first argument to loop is not needed and we can erase it, substituting b_0 for b_1 everywhere in the program. This yields:

```
pow(b0,e0):
  r0 <- 1
  goto loop(e0,r0)

loop(e1,r1):
  if (e1 > 0)
    then body()
    else done()

body():
  r3 <- r1 * b0
  e3 <- e1 - 1
  goto loop(e3,r3)

done():
  return r1
```

It is easy to check this is still in SSA form. The remaining arguments to loop are all different, however (e_0 and e_3 for e_1 and r_0 and r_2 for r_1), so we cannot optimize further.

This code is now in *minimal SSA form* in the sense that we cannot remove any label arguments by purely syntactic considerations.

The general case for this optimization is as follows: assume we have a parameterized label $l(\ldots, x_i, \ldots)$: where all gotos targeting l have the form

- goto $l(\ldots, x_i, \ldots)$ (for the same generation i) or
- goto $l(\ldots, x_k, \ldots)$ (all at the same generation k).

Then the x_i argument to l is redundant, and x_i can be replaced by x_k everywhere in the program.

6 ϕ Functions

Let's take a look at the pre-minimization version of our program again. Every use of a parameterized label has been labeled with an uppercase letter:

```
pow(b0,e0):                 // A
  r0 <- 1
  goto loop(b0,e0,r0)       // B

loop(b1,e1,r1):             // C
  if (e1 > 0)
    then body(b1,e1,r1)     // D
    else done(b1,e1,r1)     // E

body(b2,e2,r2):             // F
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop(b2,e3,r3)       // G

done(b3,e4,r4):
  return r4
```

The information we need to minimize this SSA program is, unfortunately, rather scattered about. In order to check whether we can remove the b_1 argument to loop, we have to check lines `B`, `C`, and `F`. But this is just a problem of how information is organized in the program: *first*, we have to look at a line (example: `E`), *then* we can see which label we're jumping to (example: goto), *then* we can look and see which source temps (example: b_1, e_1, r_1) will get substituted for which destination temps (example: b_3, e_4, r_4).

Put differently, the SSA part of the program is representable as a bunch of facts of the form jump($line, label, src, dst$), and in the example program above these facts are arranged in the following order:

```
jump(B, loop, b0, b1)    jump(E, done, b1, b3)
jump(B, loop, e0, e1)    jump(E, done, e1, e4)
jump(B, loop, r0, r1)    jump(E, done, r1, r4)
jump(D, body, b1, b2)    jump(G, loop, b2, b1)
jump(D, body, e1, e2)    jump(G, loop, e3, e1)
jump(D, body, r1, r2)    jump(G, loop, r3, r1)
```

But to perform SSA minimization, we want to arrange all these facts in a different way. Specifically, we want to gather all the facts with the same parameterized label and the same destination together, because if there is only one src that is associated with a given dst, or if there are two sources but one of them is equal to dst, then the parameter is unnecessary.

Rearranged for minimization, our program facts look like this, and it is immediately apparent that we can get rid of all the labels to body and done.

```
jump(B, loop, b0, b1)    jump(D, body, b1, b2)  <-- unneeded, b1=b2
jump(G, loop, b2, b1)    jump(D, body, e1, e2)  <-- unneeded, e1=e2
jump(B, loop, e0, e1)    jump(D, body, r1, r2)  <-- unneeded, r1=r2
jump(G, loop, e3, e1)    jump(E, done, b1, b3)  <-- unneeded, b1=b3
jump(B, loop, r0, r1)    jump(E, done, e1, e4)  <-- unneeded, e1=e4
jump(G, loop, r3, r1)    jump(E, done, r1, r4)  <-- unneeded, r1=r4
```

(Observe that it is *not* immediately apparent that we can get rid of b_1, because there are two sources b_0 and b_2 and neither one is equal to b_1. We only learn that we can get rid of the parameter b_1 after we perform the substitution of b_1 for b_2.)

What would the program look like if we presented it in a way that made minimization easier? We would need, associated with every parameter and every parameterized label, a list of the the lines that might jump to the parameterized label and the temp that should get substituted for the parameter when we jump from that line. The loop block would look like this:

```
loop:
  b1 <- b0 if coming here from line B,
        b2 if coming here from line G
  e1 <- e0 if coming here from line B,
        e3 if coming here from line G
  r1 <- e0 if coming here from line B,
        e3 if coming here from line G
```

This organization is actually the traditional way of presenting SSA form, except that SSA is usually presented in a more compact form called ϕ-functions. The idea is the same, but we don't mention call sites explicitly, instead we say $b_1 \leftarrow \phi(b_0, b_2)$ to represent that b_1 should, at the beginning of the loop block, be assigned to either b_0 or b_2 (whichever one we most recently wrote to). Applied to our pre-minimization example program, we get this:

```
pow(b0,e0):
  r0 <- 1
  goto loop

loop:
  b1 <- phi(b0,b2)
  e1 <- phi(e0,e3)
  r1 <- phi(e0,e3)
  if (e1 > 0) then body else done
```

```
body:
  b2 <- phi(b1)
  e2 <- phi(e1)
  r2 <- phi(r1)
  r3 <- r2 * b2
  e3 <- e2 - 1
  goto loop

done:
  b3 <- phi(b1)
  e4 <- phi(e1)
  r4 <- phi(e1)
  return r4
```

With this form, we can state the algorithm for SSA minimization described by Aycock and Horspool [AH00] the way they described it. It's quite simple: repeatedly delete ϕ-functions of the form

$$t_i = \phi(t_{x_1}, t_{x_2}, \ldots t_{x_k})$$

whenever all the x_i are either i or j. After minimization, our ϕ-function SSA looks like this:

```
pow(b0,e0):
  r0 <- 1
  goto loop

loop:
  e1 <- phi(e0,e3)
  r1 <- phi(r0,r3)
  if (e1 > 0) then body else done

body:
  r3 <- r1 * b0
  e3 <- e1 - 1
  goto loop

done:
  return r1
```

The only remaining ϕ-functions correspond to the two parameters remaining in our functional SSA program loop(e_1, r_1).

In conclusion, ϕ-function SSA is important for a number of reasons:

- It organizes information about parameters in a way that makes SSA minimization easier.
- It can be easier to understand some of the SSA-based optimizations that we'll talk about later in terms of ϕ-function SSA.
- It is the standard way of presenting SSA. If you want to read the textbook or any paper presenting SSA, you'll need to understand this form. This includes the Aycock and Horspool paper [AH00], which discusses useful implementation details and optimizations of the algorithm described here.

7 Assembly Code Generation from SSA Form

Of course, actual assembly code does not allow parameterized labels. To recover lower level code, we need to implement labeled jumps by moves followed by plain jumps. We show this again on the first example, with functional SSA and the left and the de-SSA form on the right.

```
pow(b0,e0):
  r0 <- 1
  goto loop(e0,r0)


loop(e1,r1):
  if (e1 > 0)
    then body()
    else done()

body():
  r3 <- r1 * b0
  e3 <- e1 - 1
  goto loop(e3,r3)


done():
  return r1
```

```
pow(b0,e0):
  r0 <- 1
  e1 <- e0
  r1 <- r0
  goto loop

loop:
  if (e1 > 0)
    then body
    else done

body:
  r3 <- r1 * b0
  e3 <- e1 - 1
  e1 <- e3
  r1 <- r3
  goto loop

done:
  return r1
```

In some cases of conditional jumps, there may be no natural place for the additional move instructions. This can be addressed by switching to an extended basic

block format, or by adding a new basic block that performs the moves required by SSA. Either way, we retain here the parameters at the function boundary; we will talk about the implementation of function calls in a later lecture.

The new form on the right is, of course, no longer in SSA form. Therefore one cannot apply any SSA-based optimization. Conversion out of SSA should therefore be one of the last steps before code emission. At this point register allocation, possibly with register coalescing, can do a good job of eliminating redundant moves.

8 Conclusion

Static Single Assignment (SSA) form is a quasi-functional form of abstract machine code, where variable assignments are variable bindings, and jumps are tail calls. It was devised by Cytron et al. [CFR+89] and simplifies many program analyses and optimization. Of course, you have to make sure that program transformations maintain the property. The particular algorithm for conversion into SSA form we describe here is to due Aycock and Horspool [AH00]. A final note about Aycock and Horspool's algorithm: it works for arbitrary SSA programs, but it only produces the *minimal* SSA for some programs. (Programs for which Aycock and Horspool's algorithm works are called *reducible*. All C0 programs are reducible, so the algorithm will always find the minimal SSA if you use it in your compiler.)

Hack has shown that programs in SSA form generate chordal interference graphs which means register allocation by graph coloring is particularly efficient [Hac07]. For further reading and some different algorithms related to SSA, you can also consult the Chapter 19 of the textbook [App98].

Questions

1. Can you think of an example of minimal SSA that nevertheless has redundant label arguments?

2. Can you think of situations where the control flow graph for a conditional does not have a subsequent basic block with two incoming control flow edges?

3. Give an example of a program with a non-reducible control flow graph where Aycock and Horspool's algorithm still finds the minimal SSA form.

4. Give an example of a program with a non-reducible control flow graph where Aycock and Horspool's algorithm fails to find the minimal SSA form. What is the minimal SSA form for that graph?

References

[AH00] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In D. Watt, editor, *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, pages 110–124. Springer LNCS 1781, 2000.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual Symposium on Principles of Programming Languages (POPL 1989)*, pages 25–35, Austin, Texas, January 1989. ACM Press.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.