Lecture Notes on
Static Single Assignment Form

15-411: Compiler Design
Frank Pfenning

Lecture 6
September 12, 2013

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

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

we can apply *constant propagation* of 0 to the condition 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 Lecture 5 to determine whether this is the case.

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 the 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

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, where we accept the pattern

$$\begin{array}{l}\mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l_1\\ \mathsf{goto}\ l_2\end{array}$$

at the end of a basic block. 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 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 explicity 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.

```
pow(b,e):
  r <- 1
loop:
  if (e <= 0) goto done
  r <- r * b
  e <- e - 1
  goto loop
done:
  return r
```

We can transform this into basic blocks, except that we take a small shortcut with the conditional branch by not following it with an explicit goto to save on space.

```
pow(b,e):
  r <- 1
  goto loop
loop:
  if (e <= 0) goto done
  r <- r * b
  e <- e - 1
  goto loop
done:
  return r
```

Now we note that 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 parameterize labels (the jump targets) with the variables that are live in the block that follows.[1] Labels l occuring as targets in goto l or if $(-)$ goto l are then given matching arguments.

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

loop(b,e,r):
  if (e <= 0) goto done(r)
  r <- r * b
  e <- e - 1
  goto loop(b,e,r)

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

done(r):
  return r
```

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

[1] One can also safely, but redundantly approximate this by using *all* variables.

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

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

------------------- b/1, e/2, r/2

done(r):
  return r
```

Completing the conversion with the last block, we obtain:

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

loop(b1,e1,r1):
  if (e1 <= 0) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

done(r3):
  return r3
```

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), and done(r_3). The operational reading of this program should be evident. For example, if we reach goto loop(b_0, e_0, r_0) we pass the current values of b_0, e_0 and r_0 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) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

done(r3):
  return r3
```

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

and loop(b1,e1,r1) =
    if e1 <= 0 then done(r1)
    else let r2 = r1 * b1
         and e2 = e1 - 1
          in loop(b1,e2,r2)

and done(r3) =
      r3
```

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) goto done(r1)
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)
```

```
done(r3):
  return r3
```

For example, done(r_3) is only targeted from one location, with a goto done(r_1). There is no need to pass r_1 and assign its value to r_3. We can instead remove this argument from the label done and substitute r_1 for r_3. This yields:

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

loop(b1,e1,r1):
  if (e1 <= 0) goto done()
  r2 <- r1 * b1
  e2 <- e1 - 1
  goto loop(b1,e2,r2)

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,r2)

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. This yields:

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

loop(e1,r1):
  if (e1 <= 0) goto done()
  r2 <- r1 * b0
  e2 <- e1 - 1
  goto loop(e2,r2)

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

6 Extended Basic Blocks and Conditionals

As a special case of the rule at the end of the last section, we see that if a label is the target of exactly one jump, then this condition is automatically satisfied for all of its parameters. This was the case for the label 'done' in our example. In such a case, *all* parameters of this label can be removed.

We can then go a step further and not generate parameters to labels that are targeted only once. An *extended basic block* is a collection of basic blocks with one label at the beginning (that may be the target of multiple jumps) and internal labels, each of which is the target of only one internal jump and no external jumps.

When converting to SSA form, we can treat extended basic blocks as a single unit, since we do not have to create fresh parameterized labels within them. We have already applied this idea tacitly, because in our exponential function, strictly speaking, the loop should be decomposed into basic blocks as

```
loop:
  if (e <= 0) goto done
  goto body
body:
  r <- r * b
  e <- e - 1
  goto loop
```

However, the label 'body' is targeted only by one jump, so we contracted the two instructions. The optimization discussed above is a post-hoc justification for this.

Next we consider conditionals as a new language feature. We change our program to a "fast" power function which exploits the equations $b^{2e} = (b * b)^e$ and $b^{2e+1} = b * (b * b)^e$. The C0 program is on the left, its abstract assembly form on the right.

```
int fastpow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    // r * b^e remains invariant
    {
      if (e % 2 != 0)
        r = r * b;
      b = b * b;
      e = e / 2;
    }
  return r;
}
```

```
fastpow(b,e):
  r <- 1
  goto loop
loop:
  if (e <= 0) goto done
  t0 <- e % 2
  if (t0 == 0) goto next
  r <- r * b
  goto next
next:
  b <- b * b
  e <- e / 2
  goto loop
done:
  return r
```

We see that the compiling the conditional creates another situation where we have one label (next) is the target of two jumps. This is often the case for conditionals, because the control flow graph has edges from each branch of the conditional to the statement following the conditional.

Next, we parameterize labels that are the target of more than one jump with the variables live at that program point.

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

loop(b,e,r):
  if (e <= 0) goto done
  t0 <- e % 2
  if (t0 == 0) goto next(b,e,r)
  r <- r * b
  goto next(b,e,r)

next(b,e,r):
  b <- b * b
  e <- e / 2
  goto loop(b,e,r)

done:
  return r
```

Now we convert to SSA form by generating multiple generations of variables, as in the previous example. Make sure you understand the process on this code.

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

loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(b1,e1,r1)
  r2 <- r1 * b1
  goto next(b1,e1,r2)

next(b2,e2,r3):
  b3 <- b2 * b2
  e3 <- e2 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

Next we minimize. We see the following parameterized labels and labeled jumps:

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

goto loop(b0,e0,r1)
goto loop(b3,e3,r3)

next(b2,e2,r3):

goto next(b1,e1,r1)
goto next(b1,e1,r2)
```

We see that all arguments to loop are necessary, but that the b and e arguments in the calls to next are the same and can be eliminated (substituting b_1 for b_2 and e_1 for e_2). This yields the SSA at the top of the next page.

If we want a minimal SSA (which we should), we now need to re-examine the calls to loop, because it is possible that the substitution of b_1 for b_2 and e_1 for e_2 has unified arguments that were previously distinct. That might in turn render other parameters redundant. Here, we observe that we have already reached the minimal SSA form.

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

loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)

next(r3):
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

7 Assembly Code Generation from SSA Form

Of course, actual assembly code does not allow parametrized 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 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) goto done()
  r2 <- r1 * b0
  e2 <- e1 - 1
  goto loop(e2,r2)


done():
  return r1
```

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

loop:
  if (e1 <= 0) goto done
  r2 <- r1 * b0
  e2 <- e1 - 1
  e1 <- e2
  r1 <- r2
  goto loop

done:
  return r1
```

In some cases of conditional jumps, there may be no natural place for the additional move instructions and we may have to introduce a new jump target. This is illustrated in the next example, which continues fastpow from above. All the variable to variable moves in this program arise from resolving the labeled jump shown on their left.

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


loop(b1,e1,r1):
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)


next(r3):
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)


done:
  return r1
```

```
fastpow(b0,e0):
  r0 <- 1
  b1 <- b0
  e1 <- e0
  r1 <- r0

loop:
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next0
  r2 <- r1 * b1
  r3 <- r2
  goto next

next0:
  r3 <- r1
  goto next

next:
  b3 <- b1 * b1
  e3 <- e1 / 2
  b1 <- b3
  e1 <- e3
  r1 <- r3
  goto loop

done:
  return r1
```

Either way, we retain here the parameters at the function boundary; we will talk about how 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 Φ Functions

Our presentation of SSA using parameterized labels is not standard in the literature. Instead, the standard representation of SSA form uses so-called Φ functions. A Φ function is not actually a function, but represents the assignment of the label parameter from all the jumps that target it. In order for this to be precise, we indicate for each jump which number of argument in the Φ function this particular jump represents. This perhaps best seen in the fastpow example, with the parameterized label form on the left and Φ functions on the right.

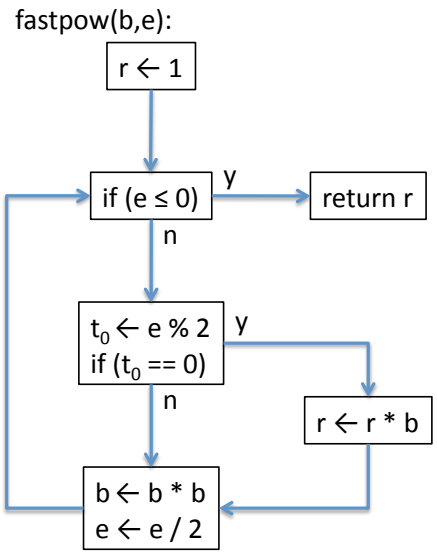
```
fastpow(b0,e0):
  r0 <- 1
  goto loop(b0,e0,r0)

loop(b1,e1,r1):


  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next(r1)
  r2 <- r1 * b1
  goto next(r2)

next(r3):

  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop(b3,e3,r3)

done:
  return r1
```

```
fastpow(b0,e0):
  r0 <- 1
  goto loop/0

loop:
  b1 <- phi(b0,b3)
  e1 <- phi(e0,e3)
  r1 <- phi(r0,r3)
  if (e1 <= 0) goto done
  t0 <- e1 % 2
  if (t0 == 0) goto next/0
  r2 <- r1 * b1
  goto next/1

next:
  r3 <- phi(r1,r2)
  b3 <- b1 * b1
  e3 <- e1 / 2
  goto loop/1

done:
  return r1
```

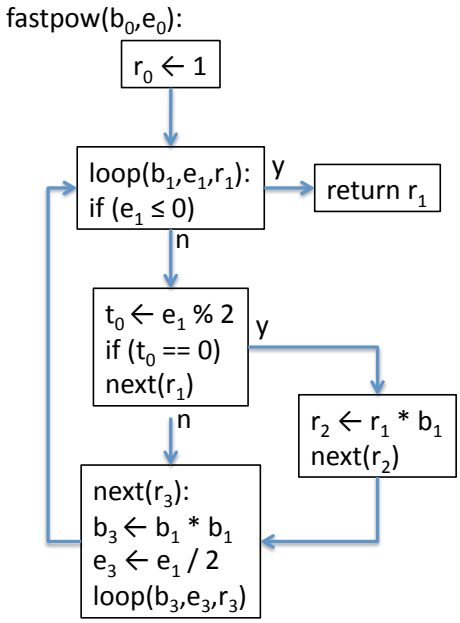
Φ functions only make sense at the beginning of blocks, and they should always have exactly as many arguments as jumps targeting the beginning of the block. Sometimes, the argument number indicators are omitted from from jumps, in which case the textual representation of the abstract assembly code does not have enough information to unambiguously determine its meaning. Then we need a global convention, such as the textually first jump supplies the first argument to the Φ functions, the second jump the second, etc.

9 Graphical Representation

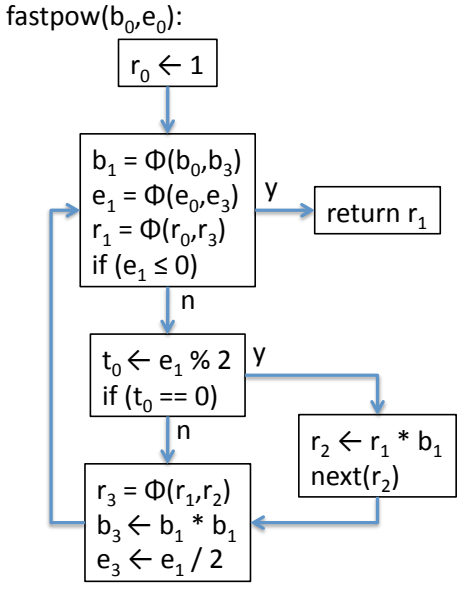
A control flow graph is often represented visually as a graph where the nodes are basic blocks and the directed edges are jump (whether conditional or not). For example, the fastpow function might be drawn as:



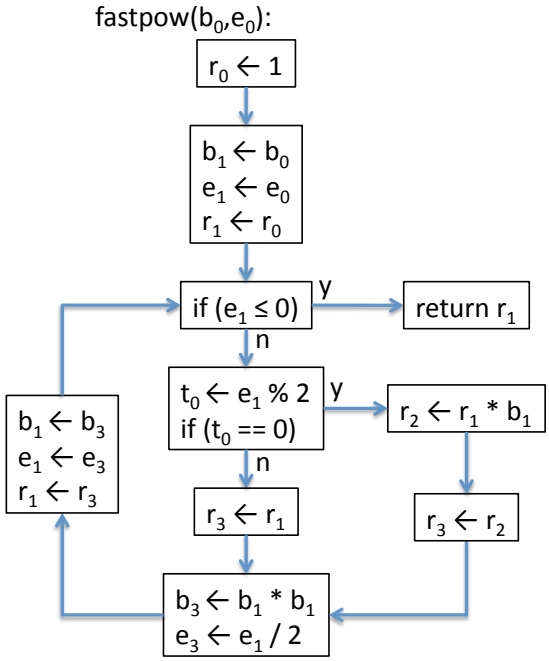
Although not commonly used, SSA form with parameterized labels might look like this:



If we use Φ-functions instead, they would be inserted at the beginning basic blocks.



When we de-SSA, the necessary register moves are added along the edges going into each node which starts with Φ-functions.



10 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]. 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?

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.