Constructing SSA the Easy Way

Michael Bebenita
University of California, Irvine

November 2, 2009

Abstract

Static Single Assignment (SSA) form has become ubiquitous in compilers as an intermediate program representation. The use of SSA simplifies many compiler optimizations and makes the life of compiler writers easier. Many techniques exist to convert programs into SSA form, many of which I find unnecessarily difficult. In this paper I will present a simple technique to convert programs in and out of SSA form.

1 Introduction

Every advanced compiler class dedicates quite a bit of time on Static Single Assignment form, or simply SSA [2]. This is because it has now become ubiquitous in compilers as an intermediate program representation. The strength of SSA lies in the fact that it simplifies many compiler optimizations by guaranteeing that variables are only assigned once. SSA also simplifies the maintenance of Use-Definition chains since each Use has only one Definition. Converting programs into SSA form is generally not possible without a little magic. Magical ϕ instructions are used to merge program states in situations where it's impossible to determine the last assignment to a variable. SSA based compilers convert programs into SSA form, perform a series of optimizations and then convert back out of SSA in later phases in the compiler. There are various ways to build SSA, some of which I find unnecessarily complicated. I'm a big fan of simplicity, so I would like to present my favorite algorithm to construct SSA and then convert back from it. This algorithm is

essentially a rehashing of Aycock's SSA construction algorithm [1] but using forwarding pointers instead.

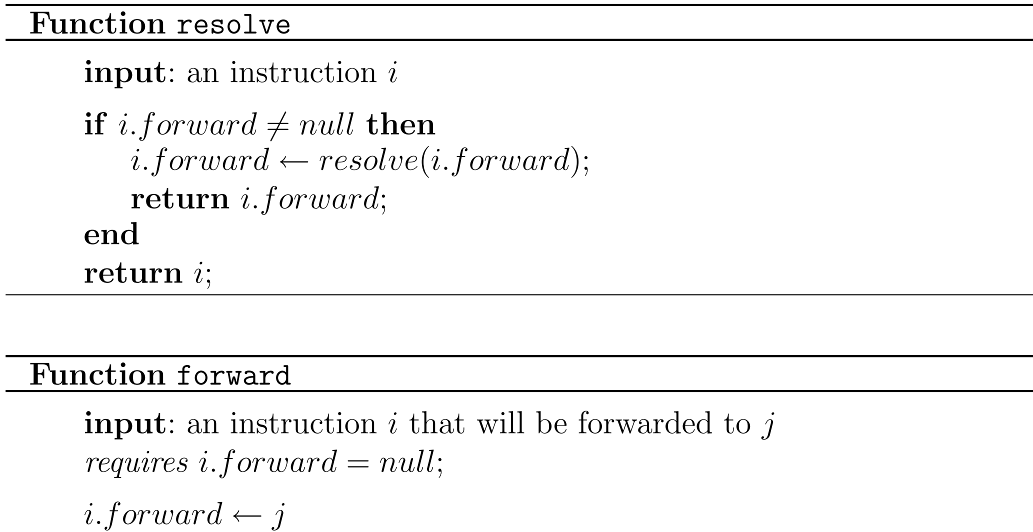
2 Intermediate Representation

In SSA form, each variable is assigned only once. Additional assignments to a variable result in redefinitions of that variable, usually notated using subscripts. Since variables can only hold one value, it is useful to eliminate the concept of a variable, and instead refer to them as values. Below an example program is shown both in the original form and its SSA equivalent. SSA values are subscripted, and each definition receives a new SSA value. The beginning of the loop body receives two sets of values: those flowing through the entry into the loop, and those looping around. To select between the two sets ϕ instructions are inserted. These abstract instructions can identify the last executed basic block, and can thus select the appropriate value.

Original Program	Program in SSA Form

$i \leftarrow 0;$
$x \leftarrow 0;$
repeat
    $z \leftarrow x;$
    $x \leftarrow y;$
    $y \leftarrow z;$
    $i \leftarrow i + 1;$
until $i < 100$;
$print(i, x, y, z);$
	$i_0 \leftarrow 0;$
	$x_0 \leftarrow 0;$
	repeat
	    $i_1 \leftarrow \phi(i_0, i_2);$
	    $x_1 \leftarrow \phi(x_0, x_2);$
	    $y_1 \leftarrow \phi(y_0, y_2);$
	    $z_1 \leftarrow \phi(z_0, z_2);$
	    $z_2 \leftarrow x_1;$
	    $x_2 \leftarrow y_1;$
	    $y_2 \leftarrow z_1;$
	    $i_2 \leftarrow i_1 + 1;$
	until $i < 100$;
	$print(i_2, x_2, y_2, z_2);$

Table 1: Original and SSA representations of an example program. Here ϕ instructions are used to select between the values flowing through the beginning of the loop, and those looping around.



Function `resolve`

input: an instruction i

if $i.forward \neq null$ **then**
  $i.forward \leftarrow resolve(i.forward)$;
  return $i.forward$;
end
return i;

Function `forward`

input: an instruction i that will be forwarded to j
requires $i.forward = null$;

$i.forward \leftarrow j$

Figure 1: Instructions must be *resolved* using the *resolve(i)* function whenever they are referenced. This allows us to replace an instruction i with any other instruction j by forwarding i to j using the *forward(i,j)* function.

3 Instruction Forwarding

Many optimizations and algorithms need to replace one SSA value with another. This requires that we know all uses of a definition. This is usually done by maintaining Definition-Use chains, and updating all uses whenever a definition is changed. Maintaining these data structures can become cumbersome. An alternative approach is to use forwarding pointers. If an SSA instruction i is replaced by another instruction j, i is forwarded to j. Each instruction has a forwarding pointer, if this pointer is not *null*, the instruction is logically replaced with the forwarded instruction. The chain of forwarding pointers must be followed whenever an instruction is accessed. The current value of an instruction i is accessed using a call to `resolve(`i`)`. The *resolve* function ensures that the entire forwarding chain is not followed every time, this is done by shortcutting to the end of the forwarding chain. Instruction forwarding has a lot of ramifications throughout the compiler. Special care must be taken to make sure that instructions are always resolved before used. The pseudo code of the *resolve* and *forward* is shown in Figure 1.

4 State Vectors

The SSA construction algorithm tracks the flow of data through the control flow graph. Usually the only dataflow that is tracked is though method local variables, but it's also possible and much more complicated to track arbitrary memory access. This is because it is difficult to observe all accesses to memory due to aliasing. This requires aggressive escape analysis to identify the dynamic scope of pointers and alias analysis to discover if multiple ways exist to access the same memory location.

Method local variables, or for that matter anything you choose to track dataflow for, can be represented as a state vector, $\{v_0, v_1, v_2, ..., v_n\}$. At each program location, a state vector indicates which SSA values are located in which local variables. State vectors can be evolved using abstract interpretation of instructions in the original program. Local variable assignments update the state vector, reads from local variables select the last written value for a given variable in the state vector. This is the way in which linear sequences of SSA instructions are generated. To generate SSA for arbitrary control flow graphs we construct SSA for each individual basic block, and then merge state vectors.

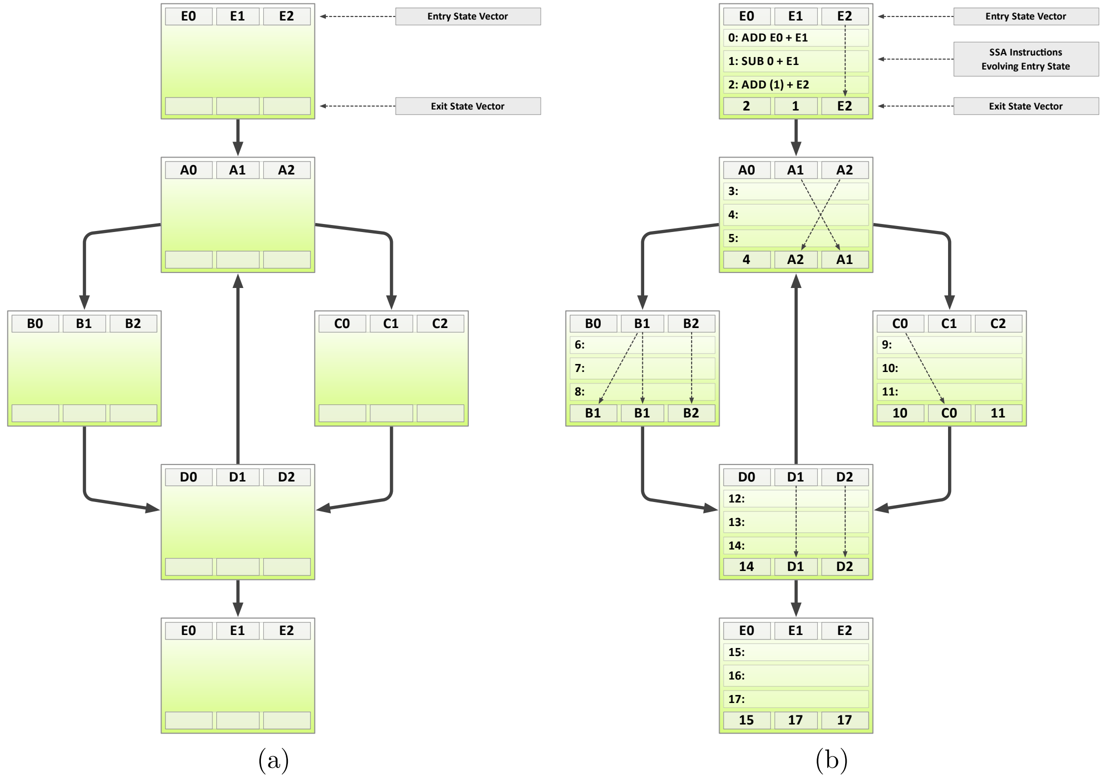


Figure 2: Phases of the SSA construction algorithm. Empty state vectors are introduced in (a), followed by construction of linear SSA for each basic block, (b).

5 SSA Construction

The SSA construction algorithm, Algorithm 3, starts off by constructing *entry* state vectors for each of the basic blocks in the control flow graph. Each of these vectors are initially filled with SSA values acting as placeholders or parameters for a given basic block (Figure 2a). SSA instructions are then constructed in a linear fashion by mutating a copy of the *entry* state vector using abstract interpretation (Figure 2b). This is what the $constructSSA(entryState, block)$ function does, it mutates the $entryState$, fills the $block$ with SSA instructions, and returns the evolved state vector at the end of the basic block. It is important to note that at this point in the

algorithm, SSA instructions only reference other SSA instructions appearing in the same basic block. The third phase of the algorithm links the *entry* and *exit* states of basic blocks. If a basic block has only one predecessor, the SSA instructions in the successor block are forwarded to the SSA instructions in the predecessor block, otherwise they are forwarded to ϕ instructions that that merge the incoming SSA instructions from all the predecessor basic blocks, Figure 5. The last phase of the algorithm optimizes away ϕ instructions whose operands are all the same. Instruction forwarding and resolution is what really makes this algorithm tick, and in my view it's quite elegant.

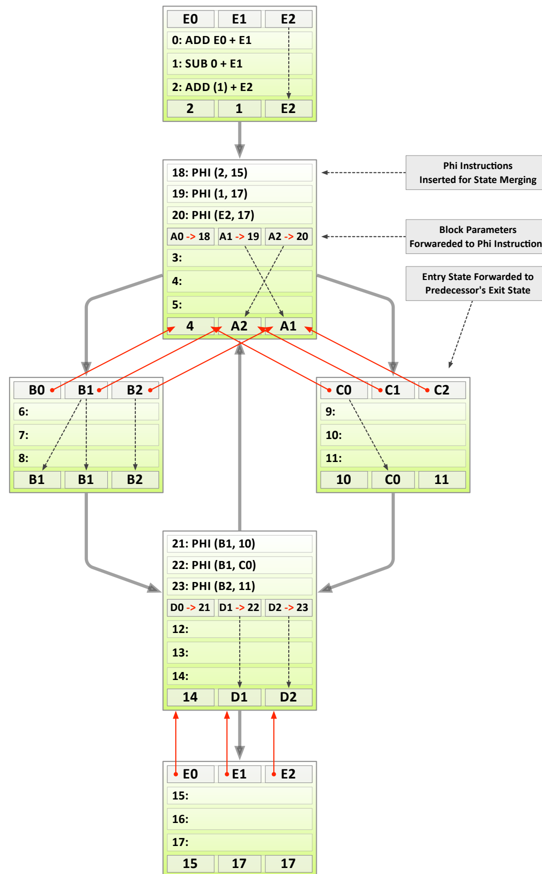


Figure 3: Entry and exit states are merged. Each block's entry state is forwarded to the predecessor's exit state, or to ϕ functions that merge all predecessor's states if more than one exists.

Algorithm 3: SSA Construction

```
input: a set of basic blocks blocks

↪ fill basic blocks with SSA instructions
for block ∈ blocks do
    entryState ← createStateVector()
    block.entryState ← entryState;
    exitState ← constructSSA(entryState, block);
    block.exitState ← exitState;
end

↪ link exit and entry states
for block ∈ blocks do
    for v_i ∈ block.entryState do
        values ← {};
        for predecessor ∈ block.predecessors do
            values.append(predecessor.exitState[i]);
        end
        if values.length = 1 then
            forward(v_i, values.first())
        else if values.length > 1 then
            phi ← createPhi(values);
            forward(v_i, phi);
            block.prepend(phi);
        end
    end
end

↪ simplify phis
for block ∈ blocks do
    for phi ∈ block do
        if phi.areAllResolvedOperandsEqual() then
            forward(phi, phi.firstResolvedOperand());
            block.remove(phi);
        end
    end
end
```

$i_0 \leftarrow 0;$
$x_0 \leftarrow 0;$
repeat
  $i_1 \leftarrow \phi(i_0, i_2);$
  $x_1 \leftarrow \phi(x_0, y_1);$
  $y_1 \leftarrow \phi(y_0, z_1);$
  $z_1 \leftarrow \phi(z_0, x_1);$
  $i_2 \leftarrow i_1 + 1;$
until $i < 100$;
$print(i_2, y_1, z_1, x_1);$

Figure 4: Copy propagation applied to the example in Table 1. A cyclic dependency develops in the ϕ instructions due to the parallel move set $\{x_1 \leftarrow y_1, y_1 \leftarrow z_1, z_1 \leftarrow x_1\}$.

6 SSA Deconstruction

Converting out of SSA form requires the introduction of moves into the predecessor blocks of ϕ instructions. At this point, variables are reintroduced into the program, every SSA value receives a variable. In order to eliminate ϕ instructions, moves are introduced in predecessor blocks, Figure 6. The semantics of ϕ instructions require that they are executed in parallel. Therefore the moves that are pushed into predecessor blocks must all execute in parallel. Because of SSA value dependencies, a topological ordering must be computed to ensure the parallel evaluation of moves executed in a sequential way. Cyclic dependencies may also occur after copy propagation, as is the case if we apply copy propagation to the example in Table 1. A cyclic dependencies develops in the loop body because of the parallel move set $\{x_1 \leftarrow y_1, y_1 \leftarrow z_1, z_1 \leftarrow x_1\}$. To break this cycle we must introduce a temporary variable as show in a more general case in Figure 6a and Figure 6b.

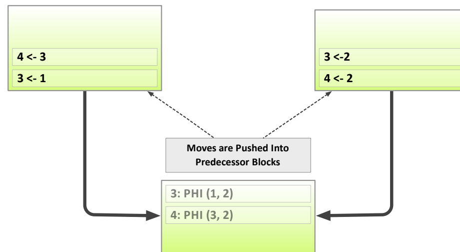


Figure 5: Eliminating ϕ instructions by introducing moves in predecessor blocks. The semantics of ϕ instructions require that they are all executed in parallel, therefore dependencies must be respected and a topological order must be computed. Here, in the left predecessor block we must make sure the variable 3 is not overwritten before it's assigned to variable 4.

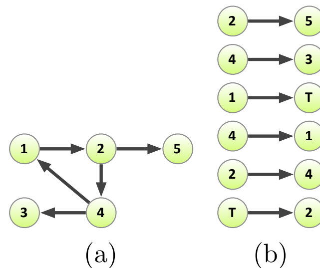


Figure 6: Cyclic dependencies and topological sorting of parallel moves. A graph of moves is show in (a), where each edge represents a move instruction. Each node can have at most one incoming edge, and many outgoing edges. A topological ordering is computed in (b) where the cycle is also broken by introducing a temporary variable, `T`.

6.1 Critical Edges

A critical edges in a control flow graph connects a block with multiple successors to a block with multiple predecessors, as shown in Figure 6.1. In such cases, eliminating ϕ instructions by pushing moves up into a predecessor block, across a critical edge, creates side effects. Alternate control flow paths leaving the predecessor block also experience the assignments and they shouldn't. The solution is to split critical edges by inserting empty

basic blocks. This provides a safe place for assignments and side effects are prevented.

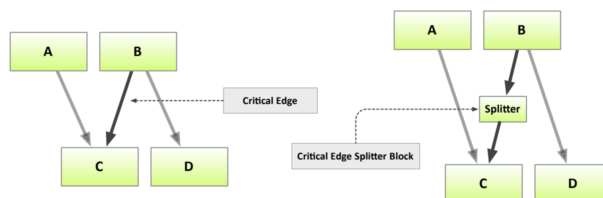


Figure 7: Critical edge splitting, an empty block is inserted in order to provide a safe place for the insertion of moves.

References

[1] J. Aycock and N. Horspool. Simple generation of static single-assignment form. *Lecture Notes in Computer Science*, pages 110–124, 2000.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.