Constructing SSA the Easy Way

Michael Bebenita
University of California, Irvine

January 25, 2011

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. This tutorial presents a technique to convert Java bytecode programs in and out of SSA form with relative ease.

1 SSA Form Construction

Static Single Assignment (SSA) form [1] has become ubiquitous in compilers as an intermediate program representation. SSA form is a program transformation where every variable is assigned exactly once. Its use greatly simplifies the way we reason about programs.

1.1 Prerequisites

You should have a good understanding of control flow graphs (CFGs), you should know what the following terms mean: basic block, successor, predecessor, join block, dominator, immediate dominator, reverse post-order traversal. If you do not go to Wikipedia and read about them. Also, you should be somewhats familiar with Java bytecodes. The structure of this tutorial is as follows:

1. We will represent program states using *State Vectors*, this is a general technique that can be used in many compiler dataflow algorithms.

2. We will perform *Abstract Interpretation* on SSA State Vectors in order to construct SSA form for each individual basic block in a control flow graph.

3. We will link the SSA form we've constructed for each individual block into a larger SSA control flow graph.

4. We will briefly discuss the dificulties when converting back from SSA form.

1.2 State Vectors

At each program point p we define the program's state at that point as s_p, where s_p is the state vector $\langle s_{pc} : l_1, l_2, ..., l_n \mid k_1, k_2, ..., k_m \rangle$. The Java bytecode position s_{pc} followed by a list of n local $(l_1, l_2, ..., l_n)$ and m stack $(k_1, k_2, ..., k_m)$ SSA values. For simplicity we make no distinction between stack and local values and simply represent the frame state vector as a list of n SSA values $\langle s_{pc} : v_1, v_2, ..., v_n \rangle$.

1.2.1 SSA Values

An SSA value is the name of a variable (value) definition. It may be a constant, a parameter, or a function of other SSA values. At each program location, a state vector indicates which SSA values are located in which local variables or stack slots. Using abstract interpretation, SSA state vectors can be mutated to reflect the flow of data through a sequence of bytecode operations. More formally, each bytecode operation has a function of the form $f_{op}(s_p) \Rightarrow (s'_p, v)$ that returns s'_p, a mutated copy of the input program state s_p and the SSA value it produced (v). The listing in Figure 1 shows three example bytecode operation functions that mutate state vectors: `LOAD_X` pushes the x^{th} local variable slot on the stack, `STORE_X` stores the top of the stack in the x^{th} local variable slot, and `ADD` creates an $add()$ SSA value of the top two SSA values on the stack and pushes it back onto the stack. One aspect worth noting is that stack load/store operations are eliminated during SSA form construction as a result of the implicit copy propagation that occurs during abstract interpretation.

Operation	Function
`LOAD_X`	$\langle s_{pc} : ..., v_x, ... \rangle \Rightarrow \langle s_{pc+1} : ..., v_x, ... \mid v_x \rangle$
`STORE_X`	$\langle s_{pc} : ..., v_x, ... \mid \mathtt{V} \rangle \Rightarrow \langle s_{pc+1} : ..., \mathtt{V}, ... \rangle$
`ADD`	$\langle s_{pc} : ... \mid \mathtt{A}, \mathtt{B} \rangle \Rightarrow \langle s_{pc+1} : ... \mid add(\mathtt{A}, \mathtt{B}) \rangle$

Figure 1: Three examples of Java bytecodes that operate on SSA state vectors.

PC	Operation	State
		Entry State : $\langle 0 : v_a, v_b, v_c \rangle$
0	`ILOAD_0`	$\langle 0 : v_a, v_b, v_c \mid v_a \rangle$
1	`ILOAD_1`	$\langle 1 : v_a, v_b, v_c \mid v_a, v_b \rangle$
2	`IADD`	$\langle 2 : v_a, v_b, v_c \mid add(v_a, v_b) \rangle$
3	`ISTORE_2`	$\langle 3 : v_a, v_b, add(v_a, v_b) \rangle$
4	`ICONST_3`	$\langle 4 : v_a, v_b, add(v_a, v_b) \mid 3 \rangle$
5	`ISTORE_1`	$\langle 5 : v_a, 3, add(v_a, v_b) \rangle$
6	`ILOAD_1`	$\langle 6 : v_a, 3, add(v_a, v_b) \mid 3 \rangle$
7	`ILOAD_2`	$\langle 7 : v_a, 3, add(v_a, v_b) \mid 3, add(v_a, v_b) \rangle$
8	`IMUL`	$\langle 8 : v_a, 3, add(v_a, v_b) \mid mul(3, add(v_a, v_b)) \rangle$
9	`ISTORE_1`	$\langle 9 : v_a, mul(3, add(v_a, v_b)), add(v_a, v_b) \rangle$
		Exit State : $\langle 9 : v_a, 3, add(v_a, v_b), v_d \rangle$

Figure 2: Abstract Interpretation

1.2.2 Example

The abstract interpretation sequence of the following short program (left) listing and its SSA form counterpart (right) is presented in Figure 2.

$c \leftarrow a + b;$	$c_0 \leftarrow a_0 + b_0;$
$b \leftarrow 3;$	$b_1 \leftarrow 3;$
$b \leftarrow b \times c;$	$b_2 \leftarrow b_1 \times c_0;$

Figure 2 shows how an SSA state vector is mutated using abstract interpretation by successively applying a sequence of bytecode operation functions. The bytecode sequence is the equivalent of the above program listing.

1.3 SSA Control Flow Graph

Abstract interpretation allows us to generate SSA form for linear sequences of program code (or basic blocks). In order to model arbitrary control flow we must link basic blocks together and merge state vectors at program merge points (Figure 3). In fact, our algorithm uses abstract interpretation to fill in the SSA IR (intermediate representation) for each basic block independently. And then, with the help of SSA Value Forwarding (Section 1.3.1), it joins basic blocks and inserts ϕ (Phi) functions to merge control flow at program merge points. Phi functions are of the form:

$$\begin{aligned}
&s'' = \phi(s, s', ...) \Rightarrow \\
&s'' = \phi(\langle s_{pc} : v_1, v_2, ..., v_n\rangle, \langle s'_{pc} : v'_1, v'_2, ..., v'_n\rangle, ...) \Rightarrow \\
&s''_s = \langle s''_{pc} : \phi(v_1, v'_1, ...), \phi(v_2, v'_2, ...), ..., \phi(v_n, v'_n, ...)\rangle
\end{aligned}$$

The function, $\phi(v_n, v'_n, ...)$ selects one of the operands depending on where control flow arrives to the ϕ function from. It is important to note that ϕ functions are applied to state vectors, and thus, the selection of all individual components must happen atomically with respect to the state vectors. Cycles may appear between the operands of ϕ functions due to copy propagation and it is important to consider the ϕ operand selection as though it happens in parallel.

1.3.1 SSA Value Forwarding

Many optimizations and algorithms need to replace one SSA value with another. A naive implementation would require that all uses of an SSA value definition are maintained in an auxiliary Def-Use chain data structure. Whenever an SSA value is replaced with another, all uses of the original value are updated to refer to the updated value. Unfortunately, maintaining Def-Use chains can become cumbersome and inefficient, as it requires that all uses be updated whenever an SSA value is replaced with another. Instead, we lazily evaluate each use of an SSA value by following a chain of forwarding pointers. Each SSA value has a forwarding pointer. If this pointer is set, then the SSA value is logically replaced with the value it is being forwarded to. To do this we use the forwarding function $forward(i, j)$ to replace i with j $(i \rightarrow j)$. This way, whenever an SSA value is replaced by another, the old SSA value is simply forwarded to the new one by linking the two with a forwarding pointer.

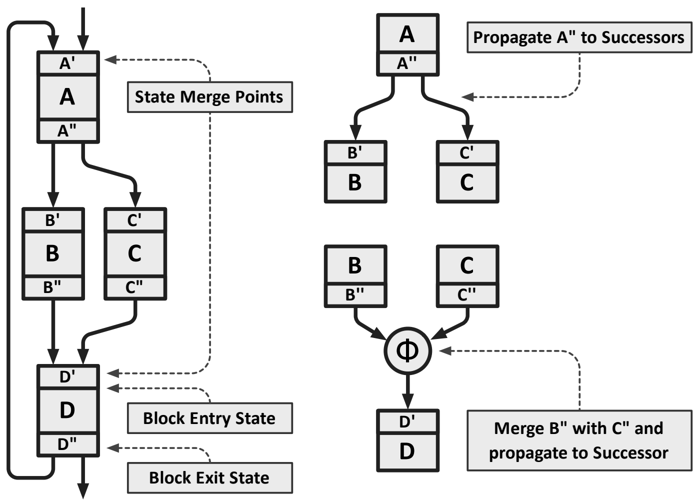

Figure 3: Entry and exit state vectors are merged. Each block's exit state is either directly propagated to its successor block, or propagated through a state vector merge function, namely the ϕ function.

Therefore, the identity of an SSA value at a Use site is defined by the value resolution function:

$$resolve(x) = \begin{cases} x & \text{if not forwarded} \\ resolve(x.forward) & \text{otherwise} \end{cases}$$

The chain of forwarding pointers must be followed whenever an instruction is accessed. The current "real" value of an SSA value i is given only by $resolve(i)$.

The resolution function also performs pointer swizzling[1] in order to limit the value forwarding chain to at most one link. If a value i is forwarded to j ($i \rightarrow j$), and j is then forwarded to k ($j \rightarrow k$), then whenever i is resolved it is automatically forwarded to $resolve(i)$. This effectively caches value resolution and is possible for the following reasons:

- An SSA value cannot be forwarded to itself ($i \rightarrow i$). This is to prevent cycles in the forwarding chain. Although the SSA value graph may contain cycles, the forwarding chain is always acyclic.

[1]Pointer swizzling is similar to path compression in the Union-Find data structure.

- If a value i is forwarded to j ($i \to j$), then i cannot be forwarded again, since i is immediately replaced with j ($i \neq resolve(i)$). Therefore, there is no reason to ever invalidate the value resolution cache.

1.4 Basic Block Construction & State Vector Linking

The SSA form construction algorithm (Algorithm 1) operates in three phases:

Phase 1: SSA Value Graph Construction The first phase of the algorithm constructs an SSA value graph for each of the basic blocks in the CFG using abstract interpretation. The initial SSA entry state of a basic block is filled with SSA parameter values. SSA parameter values are placeholder values representing the incoming values into a basic block. For each bytecode operation the algorithm then constructs SSA values by successively applying each bytecode's operation function. The application of the operation function mutates the state vector and creates a new SSA value which is then added to a sequence of SSA instructions/values in the basic block. The SSA exit state for the basic block is recorded after the application of the last bytecode operation function. This phase of the algorithm constructs an acyclic SSA graph for each basic block rooted in SSA parameter values.

Phase 2: Basic Block Linking & State Forwarding $\forall\ b \in B : b.exitState \approx successor(b).entryState$, where B is the set of basic blocks in a CFG. This means that for all blocks, the *exitState* can be propagated to all successor blocks. This is done by forwarding the entry state of each block to its predecessor's exit state (we define state forwarding as: $forward(p_s, p'_s) \Rightarrow forward(v_1, v'_1), ..., forward(v_n, v'_n)$).

Each basic block may have one or more predecessor blocks. If a block has only one predecessor, its entry state is forwarded to the exit state of the predecessor block. If a block has two or more predecessors, its entry state is forwarded to a ϕ function merging the exit states of predecessor blocks. At this point, the block's entry state, which used to contain SSA parameter values, is forwarded and swizzled away to the forwarded values. Since the ϕ function is distributed across each component of the state vector, we avoid the insertion of ϕ functions whenever we notice that it is merging equal SSA values, and simply forward to the operand instead. This optimization is sensitive to the order in which we link basic blocks. The pseudocode in Algorithm 1 purposely fails to indicate a block linking order because it is not necessary for correctness. However, if we process blocks in reverse postorder, we can drastically reduce the number of inserted ϕ functions. This is because

Algorithm 1: Static SSA Form Construction

```
input  : A CFG of basic $blocks$ that are linked and contain Java bytecode
         operations but are not in SSA form yet

↪ Phase 1: Perform abstract interpretation on all basic blocks and fill each
block with SSA instructions/values.

for $block \in blocks$ do
    $state \leftarrow createStateVector()$;
    $block.entryState \leftarrow state$;
    for $bytecode \in block.bytecodes$ do
        $(state, v) \leftarrow f_{bytecode}(state)$;
        $block.appendInstruction(v)$;
    $block.exitState \leftarrow state$;

↪ Phase 2: Link basic block by forwarding entry states to predecessor's exit
states.

for $block \in blocks$ do
    if $isUniqueSuccessor(block)$ then
        $forward(block.entryState, predExitState(block))$;
    else
        $forward(block.entryState, \phi(predExitStates(block))$;

↪ Phase 3: Optimize $\phi$ instructions.
```

we can guarantee that a block's predecessor blocks have already been linked and their entry states forwarded, and thus we don't need to worry about inserting ϕ functions for SSA parameter values that may have propagated through predecessor blocks. This phase completes SSA construction.

Phase 3: ϕ Function Elimination The third and last phase of the algorithm simplifies the SSA graph by iteratively eliminating ϕ functions. Functions of the form $v_2 = \phi(v_1, v_1, ..., v_1)$ are forwarded to their common operand $(v_2 \rightarrow v_1)$. Functions of the form $v_2 = \phi(v_1, v_2, ..., v_2)$ are forwarded to their incoming operand $(v_2 \rightarrow v_1)$. This is a fixed-point algorithm which eliminates ϕ functions until no further progress is made.

2 SSA Deconstruction

Converting out of SSA form requires the introduction of move instructions into the predecessor blocks of blocks containing ϕ instructions (Figure 4). At this point, variables are reintroduced into the program for every SSA value. The semantics of ϕ instructions require that they execute in parallel. Therefore the move instructions that are pushed into predecessor blocks must all execute in parallel as well. Because of SSA value dependencies, a topological ordering must be computed to ensure the parallel evaluation semantics of moves are preserved when executing them in a sequential way. Cyclic dependencies may also occur after copy propagation, to break cycles we must introduce temporary variables, or use exchange operations (Figure 5).

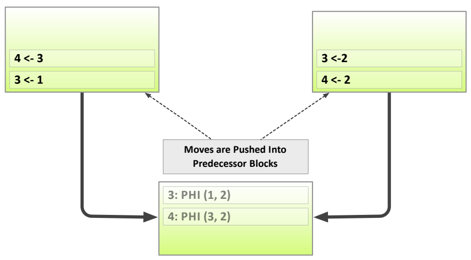

Figure 4: Eliminating ϕ instructions by introducing moves in predecessor blocks. The semantics of ϕ instructions require that they are all executed in parallel, therefore dependencies must be respected and a topological order must be computed. Here, in the left predecessor block we must make sure the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

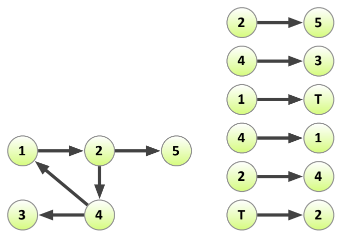

Figure 5: Cyclic dependencies and topological sorting of parallel moves. A graph of moves (left), where each edge represents a move instruction. Each node can have at most one incoming edge, and many outgoing edges. A topological ordering is computed (right) where the cycle is also broken by introducing a temporary variable, `T`.

2.1 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. In such cases, eliminating ϕ instructions by pushing moves up into a predecessor block, across a critical edge, causes 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 to be placed.

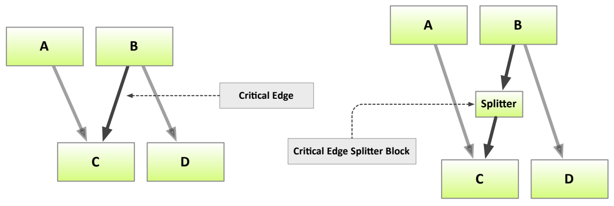

Figure 6: Critical edge splitting, an empty block is inserted in order to provide a safe place for the insertion of moves.

References

[1] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control de-

pendence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.