Merged too small paragraphs.
Put the figures closed to the text
Added many index

CHAPTER 1

Standard Construction and Destruction Algorithms

J. Singer

F. Rastello

Progress: 90%

Text and figures formating in progress

This chapter describes the standard algorithms for construction and destruction of SSA form. SSA *construction* refers to the process of translating a non-SSA program into one that satisfies the SSA constraints. In general, this transformation occurs as one of the earliest phases in the middle-end of an optimizing compiler, when the program has been converted to three-address intermediate code. SSA *destruction* is sometimes called out-of-SSA translation. This step generally takes place in an optimizing compiler after all SSA optimizations have been performed, and prior to code generation. However note that there are specialized code generation techniques that can work directly on SSA-based intermediate representations such as instruction selection (see Chapter **??**), if-conversion (see Chapter **??**), and register allocation (see Chapter **??**).

The algorithms presented in this chapter are based on material from the seminal research papers on SSA. These original algorithms are straightforward to implement and have acceptable efficiency. Therefore such algorithms are widely implemented in current compilers. Note that more efficient, albeit more complex, alternative algorithms have been devised. These are described further in Chapters **??** and **??**.

Figure 1.1 shows an example CFG program. The set of nodes is $\{r, A, B, C, D, E\}$. The set of variables is $\{x, y, tmp\}$. Note that the program shows the complete control flow structure, denoted by directed edges between the nodes. However the program only shows statements that define relevant variables, together with the unique `return` statement at the exit point of the CFG. All of the program variables are undefined on entry. On certain control flow paths, some variables may be used without being defined, e.g. x on the path $r \rightarrow A \rightarrow C$. We discuss this issue later in the chapter. We intend to use this program as a running example throughout the chapter, to demonstrate various aspects of SSA construction.

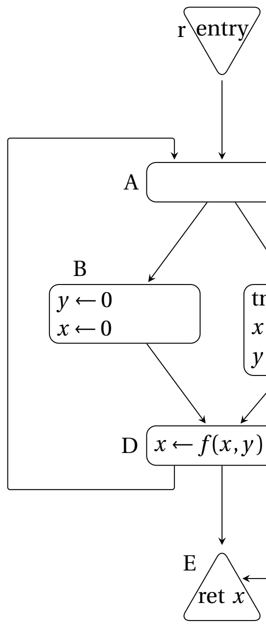


Fig. 1.1 Example control flow graph, before SSA construction occurs

1.1 Construction

The original construction algorithm for SSA form consists of two distinct phases.

1. **ϕ-function insertion** performs *live-range splitting* to ensures that any use of a given variable v is reached[1] by exactly one definition of v. The resulting live-ranges exhibit the property of having a single definition, which occurs at the beginning of each live-range.
2. **variable renaming** assigns a unique variable name to each live-range. This second phase rewrites variable names in program statements such that the program text contains only one definition of each variable, and every use refers to its corresponding unique reaching definition.

As already outlined in Chapter **??**, there are different flavors of SSA with distinct properties. In this chapter, we focus on the *minimal* SSA form.

[1] A program point p is said to be *reachable* by a definition of v, if there exists a path in the CFG from that definition to p that does not contain any other definition of v.

Join Sets and Dominance Frontiers

In order to explain how ϕ-function insertions occur, it will be helpful to review the related concepts of *join sets* and *dominance frontiers*.

For a given set of nodes S in a CFG, the join set $\mathcal{J}(S)$ is the set of *join nodes* of S, i.e. nodes in the CFG that can be reached by two (or more) distinct elements of S using disjoint paths. Join sets were introduced in Section **??**.

Let us consider some join set examples from the program in Figure 1.1.

1. $\mathcal{J}(\{B, C\}) = \{D\}$, since it is possible to get from B to D and from C to D along different, non-overlapping, paths.
2. Again, $\mathcal{J}(\{r, A, B, C, D, E\}) = \{A, D, E\}$, since the nodes A, D, and E are the only nodes with multiple predecessors in the program.

The *dominance frontier* of a node n, DF(n), is the border of the CFG region that is dominated by n. More formally,

- node a *strictly dominates* node b if a dominates b and $a \neq b$
- the set of nodes DF(a) contains all nodes n such that a dominates a predecessor of n but a does not strictly dominate n.

Note that DF is defined over individual nodes, but for simplicity of presentation, we overload it to operate over sets of nodes too, i.e. $\mathrm{DF}(S) = \bigcup_{s \in S} \mathrm{DF}(s)$. The *iterated dominance frontier* $\mathrm{DF}^+(S)$, is the limit $DF_{i \to \infty}(S)$ of the sequence:

$$\begin{aligned} \mathrm{DF}_1(S) &= \mathrm{DF}(S) \\ \mathrm{DF}_{i+1}(S) &= \mathrm{DF}(S \cup \mathrm{DF}_i(S)) \end{aligned}$$

Construction of minimal SSA requires for each variable v the insertion of ϕ-functions at $\mathcal{J}(D_v)$, where D_v is the set of nodes that contain definitions of v. The original construction algorithm for SSA form uses the iterated dominance frontier $\mathrm{DF}^+(D_v)$. This is an over-approximation of join set, since $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$, i.e. the original algorithm assumes an *implicit* definition of every variable at the entry node r.

ϕ-function Insertion

This concept of dominance frontiers naturally leads to a straightforward approach that places ϕ-functions on a per-variable basis. For a given variable v, we place ϕ-functions at the iterated dominance frontier $\mathrm{DF}^+(D_v)$ where D_v is the set of nodes containing definitions of v. This leads to the construction of SSA form that has the dominance property, i.e. where each renamed variable's definition dominates its entire live-range.

Consider again our running example from Figure 1.1. The set of nodes containing definitions of variable x is $\{B, C, D\}$. The iterated dominance frontier of this set is $\{A, D, E\}$. Hence we need to insert ϕ-functions for x at the beginning of nodes A, D, and E. Figure 1.2 shows the example CFG program with ϕ-functions for x inserted.

As far as the actual algorithm for ϕ-functions insertion is concerned, we will assume that the dominance frontier of each CFG node is pre-computed and that the iterated dominance frontier is computed just-in-time, as the algorithm proceeds. The algorithm works by inserting ϕ-functions iteratively using a worklist of definition points, and flags (to avoid multiple insertions). The corresponding pseudo-code for ϕ-function insertion is given in Algorithm 1. The worklist of nodes W is used to record definition points that the algorithm has not yet processed, i.e. it has not yet inserted ϕ-functions at their dominance frontiers. Because a ϕ-function is itself a definition, it may require further ϕ-functions to be inserted. This is the cause of node insertions into the worklist W during iterations of the inner loop in Algorithm 1. Effectively, this is just-in-time calculation of the iterated dominance frontier. The flags array F is used to avoid repeated insertion of ϕ-functions at a single node. Dominance frontiers of distinct nodes may intersect, e.g. in the example CFG in Figure 1.1, DF(B) and DF(C) both contain D. Once a ϕ-function for a particular variable has been inserted at a node, there is no need to insert another, since a single ϕ-function per variable handles all incoming definitions of that variable to that node.

Algorithm 1: Standard algorithm for inserting ϕ-functions for a variable v

```
1  begin
2      $W \leftarrow \{\}$ : set of basic blocks;
3      $F \leftarrow \{\}$ : set of basic blocks;
4      for $v$ : variable names in original program do
5          for $d \in D_v$ do
6              let $B$ be the basic block containing $d$;
7              $W \leftarrow W \cup \{B\}$;
8          while $W \neq \{\}$ do
9              remove a basic block $X$ from $W$;
10             for $Y$ : basic block $\in$ DF($X$) do
11                 if $Y \notin F$ then
12                     add $v \leftarrow \phi(...)$ at entry of $Y$;
13                     $F \leftarrow F \cup \{Y\}$;
14                     if $Y \notin D_v$ then
15                         $W \leftarrow W \cup \{Y\}$;
16 end
```

Table 1.1 gives a walk-through example of Algorithm 1. It shows the stages of execution for a single iteration of the outermost for loop, which is inserting ϕ-functions for variable x. Each row represents a single iteration of the while loop that iterates over the worklist W. The table shows the values of X, F and W at the start of each while loop iteration. At the beginning, the CFG looks like Figure 1.1. At the end, when all the ϕ-functions for x have been placed, then the CFG looks like Figure 1.2.

while loop #	X	DF(X)	F	W
-	-	-	$\{\}$	$\{B,C,D\}$
0	B	$\{D\}$	$\{D\}$	$\{C,D\}$
1	C	$\{D,E\}$	$\{D,E\}$	$\{D,E\}$
2	D	$\{E,A\}$	$\{D,E,A\}$	$\{E,A\}$
3	E	$\{\}$	$\{D,E,A\}$	$\{A\}$
4	A	$\{A\}$	$\{D,E,A\}$	$\{\}$

Table 1.1 Walk-through of placement of ϕ-functions for variable x in example CFG

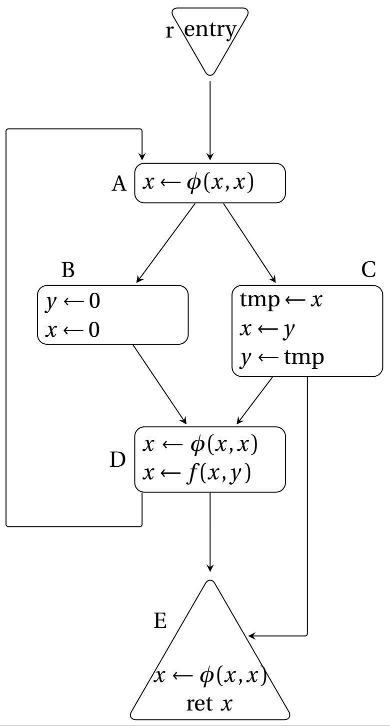


Fig. 1.2 Example control flow graph, including inserted ϕ-functions for variable x

Providing the dominator tree is given, the computation of the dominance frontier is quite straightforward. As illustrated by Figure 1.3, this can be understood using the DJ-graph notation. The skeleton of the DJ-graph is the dominator tree of the CFG that makes the D-edges. This is augmented with J-edges (join edges) that correspond to all edges of the CFG whose source does not strictly dominate its destination. A DF-edge (dominance frontier edge) is an edge whose destination is in the dominance frontier of its source. By definition, there is a DF-edge (a,b) between every CFG nodes a, b such that a dominates a predecessor of b, but does not strictly dominate b. In other-words, for each J-edge (a,b), all ancestors of a (including a) that do not strictly dominate b have b

Illustrated using the example

in their dominance frontier. For example, in Figure 1.3, (F,G) is a J-edge, so $\{(F,G),\ (E,G),\ (B,G)\}$ are DF-edges. This leads to the pseudo-code given in Algorithm 2. Since the iterated dominance frontier is simply the transitive closure of the dominance frontier, then we can define the DF$^+$-graph as the transitive closure of the DF-graph. In our example, as $\{(C,E),\ (E,G)\}$ are DF-edges, (C,G) is a DF$^+$-edge. Hence, a definition of x in C will lead to inserting ϕ-functions in E and G. We can compute the iterated dominance frontier for each variable independently, as outlined in this chapter, or 'cache' it to avoid repeated computation of the iterated dominance frontier of the same node. This leads to more sophisticated algorithms detailed in Chapter **??**.

Changed the example

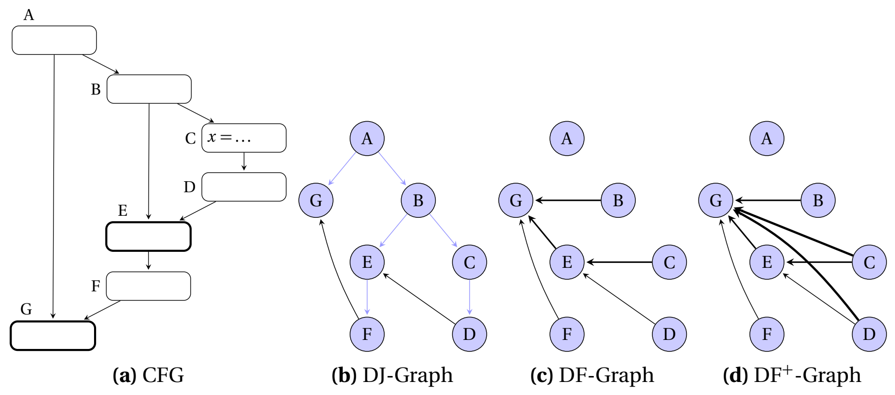


Fig. 1.3 An example CFG and it corresponding DJ-graph (D-edges are highlighted), DF-graph and DF$^+$-graph. DF$^+(C) = \{E,G\}$

Algorithm 2: Algorithm for computing the dominance frontier of each CFG node

a can be in the DF of itself

```
1 begin
2     for $(a,b) \in$ cfgEdges do
3         while $a$ does not strictly dominate $b$ do
4             $\mathrm{DF}(a) \leftarrow \mathrm{DF}(a) \cup b$;
5             $a \leftarrow \mathrm{iDom}(a)$;
6 end
```

Once ϕ-functions have been inserted using this algorithm, the program may still contain several definitions per variable, however now there is a single definition statement in the CFG that reaches each use. It is conventional to treat each variable use in a ϕ-function as if it actually occurs on the corresponding incoming edge or at the end of the corresponding predecessor node. If we follow this convention, then def-use chains are aligned with the CFG dominator tree. In other words, *the single definition that reaches each use dominates that use.*

Variable Renaming

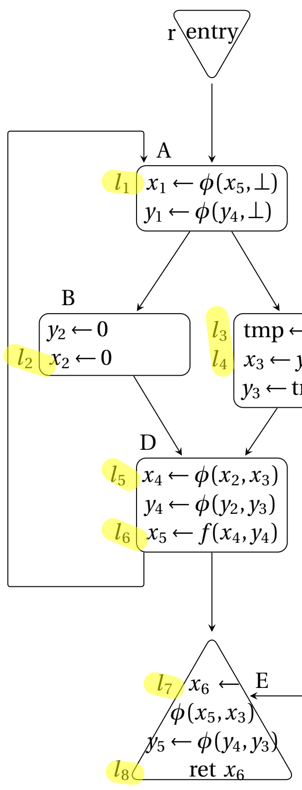
To obtain the desired property of a static single assignment per variable, it is necessary to perform variable renaming, which is the second phase in the SSA construction process. ϕ-function insertions have the effect of splitting the live-range(s) of each original variable into pieces. The variable renaming phase associates to each individual live-range a new variable name, also called a *version*. Algorithm 3 presents the pseudo-code for this process. Because of the dominance property outlined above, it is straightforward to rename variables using a depth-first traversal of the dominator tree. During the traversal, for each variable v, it is necessary to remember its unique reaching version's definition at some point p in the graph. This corresponds to the closest definition that dominates p. In Algorithm 3, we compute and cache the reaching definition for v in the per-variable slot v.reachingDef that is updated as the algorithm traverses the dominator tree of the SSA graph. This per-variable slot stores the in-scope, 'new' variable name (version) for the equivalent variable at the same point in the un-renamed program.

Algorithm 3: Renaming algorithm for second phase of SSA construction

```
begin
    /* rename variable definitions and uses to have one definition per
       variable name                                                  */
    foreach v : Variable do
        v.reachingDef ← ⊥;
    foreach BB : basic Block in depth-first search preorder traversal of the dominance tree
    do
        foreach i : instruction in linear code sequence of BB do
            foreach v : variable used as a source operand on right-hand side of
            non-$\phi$-function i do
                updateReachingDef(v, i);
                replace this use of v by v.reachingDef in i;
            foreach v : variable defined by i, which may be a $\phi$-function do
                updateReachingDef(v, i);
                create fresh variable v′;
                replace this definition of v by v′ in i;
                v′.reachingDef ← v.reachingDef ;
                v.reachingDef ← v′;
        foreach f: $\phi$-function in a successor of BB do
            foreach v : variable used as a source operand on right-hand side of f do
                updateReachingDef(v, f);
                replace this use of v by v.reachingDef in f;
end
```

The variable renaming algorithm translates our running example from Figure 1.1 into the SSA form of Figure 1.4a. The table in Figure 1.4b gives a walk-through example of Algorithm 3, only considering variable x. The labels l_i mark instructions in the program that mention x, shown in Figure 1.4a. The table records (1) when x.reachingDef is updated from x_{old} into x_{new} due to a call of updateReachingDef, and (2) when x.reachingDef is x_{old} before a definition statement, then x_{new} afterwards.

Added labels and fixed errors


r entry
A
l_1 $x_1 \leftarrow \phi(x_5,\perp)$
$y_1 \leftarrow \phi(y_4,\perp)$
B
$y_2 \leftarrow 0$
l_2 $x_2 \leftarrow 0$
C
l_3 tmp $\leftarrow x_1$
l_4 $x_3 \leftarrow y_1$
$y_3 \leftarrow$ tmp
D
l_5 $x_4 \leftarrow \phi(x_2,x_3)$
$y_4 \leftarrow \phi(y_2,y_3)$
l_6 $x_5 \leftarrow f(x_4,y_4)$
E
l_7 $x_6 \leftarrow \phi(x_5,x_3)$
$y_5 \leftarrow \phi(y_4,y_3)$
l_8 ret x_6

(a) CFG

BB	x mention	x.reachingDef
r	use at l_1	$\perp$
A	def at l_1	$\perp$ then x_1
B	def at l_2	x_1 then x_2
B	use at l_5	x_2
C	use at l_3	x_2 updated into x_1
C	def at l_4	x_1 then x_3
C	use at l_5	x_3
C	use at l_7	x_3
D	def at l_5	x_3 updated into x_1 then x_4
D	use at l_6	x_4
D	def at l_6	x_4 then x_5
D	use at l_1	x_5
D	use at l_7	x_5
E	def at l_7	x_5 then x_6
E	use at l_8	x_6

(b) Walktrough of renaming for variable x in example CFG

Fig. 1.4 SSA form of the example of Figure 1.1

The original presentation of the renaming algorithm uses a per-variable stack that stores all variable versions that dominate the current program point, rather than the slot-based approach outlined above. In the stack-based algorithm, a stack value is pushed when a variable definition is processed (while we explicitly update the reachingDef field at this point). The top stack value is peeked when a variable use is encountered (we read from the reachingDef field at this point). Multiple stack values may be popped when moving to a different node in the dominator tree (we always check whether we need to update the reachingDef field before we read from it). While the slot-based algorithm requires more memory, it can take advantage of an existing variable's working field, and be more efficient in practice.

Procedure `updateReachingDef`(v,i) Utility function for SSA renaming

```
Data: $v$ : variable from program
Data: $i$ : instruction from program
begin
    /* search through chain of definitions for v until we find the
       closest definition that dominates i, then update $v$.reachingDef
       in-place with this definition                                   */
    $r \leftarrow v$.reachingDef;
    while not ($r == \perp$ or definition($r$) dominates $i$) do
        $r \leftarrow r$.reachingDef;
    $v$.reachingDef $\leftarrow r$;
end
```

Summary

Now let us review the flavour of SSA form that this simple construction algorithm produces. We refer back to several SSA properties that were introduced in Chapter **??**.

- It is *minimal*, see Section **??**. After the ϕ-function insertion phase, but before variable renaming, the CFG contains the minimal number of inserted ϕ-functions to achieve the property that exactly one definition of each variable v reaches every point in the graph.
- It is *not pruned*, see Section **??**. Some of the inserted ϕ-functions may be dead(e.g. y_5 in Figure 1.4a), i.e. there is no explicit use of the variable subsequent to the ϕ-function.
- It is *conventional*, see Section **??**. The transformation that renames all ϕ-related variables into a unique representative name and then removes all ϕ-functions is a correct SSA-destruction algorithm.
- Finally, it has the *dominance* property, see Section **??**. Each variable use is dominated by its unique definition. This is due to the use of iterated dominance frontiers during the ϕ-placement phase, rather than join sets. Whenever the iterated dominance frontier of the set of definition points of a variable differs from its join set, then at least one program point can be reached both by r (the entry of the CFG) and one of the definition points. In other words, as in Figure 1.1, one of the uses of the ϕ-function inserted in block A for x does not have any actual reaching definition that dominates it. This corresponds to the $\perp$ value used to initialize each reachingDef slot in Algorithm 3. Actual implementation code can use a `NULL` value, create a fake undefined variable at the entry of the CFG, or create on the fly undefined pseudo-operations just before the particular use.

1.2 Destruction

SSA form is a sparse representation of program information, which enables simple, efficient code analysis and optimization. Once we have completed SSA based optimization passes, and certainly before code generation, it is necessary to eliminate ϕ-functions since these are not executable machine instructions. This elimination phase is known as *SSA destruction*.

Since freshly constructed, untransformed SSA code is conventional, its destruction is straightforward. One simply has to rename all ϕ-related variables (source and destination operands of a single ϕ-function are related) into a unique representative variable. Then each ϕ-function should have syntactically identical names for all its operands, and thus can be removed to coalesce the related live-ranges.

We refer to a set of ϕ-related variables as a ϕ-web. We recall from Chapter **??** that conventional SSA is defined as a flavor under which each ϕ-web is interference free. In particular if each ϕ-web's constituent variables have non-overlapping live-ranges then the SSA form is conventional. The discovery of ϕ-webs can be performed efficiently using the classical *union-find* algorithm with a disjoint-set data structure, which keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. The ϕ-webs discovery algorithm is presented in Algorithm 5.

Algorithm 5: The ϕ-webs discovery algorithm, based on the union-find pattern

```
1 begin
2     for each variable $v$ do
3         phiweb($v$) ← {$v$};
4     for each instruction of the form $a_{dest} = \phi(a_1, \dots, a_n)$ do
5         for each source operand $a_i$ in instruction do
6             union(phiweb($a_{dest}$), phiweb($a_i$))
7 end
```

While freshly constructed SSA code is conventional, this may not be the case after optimizations such as copy propagation have been performed. Going back to conventional SSA form implies the insertion of copies. The simplest (although not the most efficient) way to destroy non-conventional SSA form is to split all *critical edges*, and then replace ϕ-functions by copies at the end of predecessor basic blocks. A critical edge is an edge from a node with several successors to a node with several predecessors. The process of splitting an edge, say (b_1, b_2), involves replacing edge (b_1, b_2) by (i) an edge from b_1 to a freshly created basic block and by (ii) another edge from this fresh basic block to b_2. As ϕ-functions have a parallel semantic, i.e. have to be executed simultaneously not sequentially, the same holds for the corresponding copies inserted at the end of predecessor ba-

We were talkign about parallel copies without giving a definition sic blocks. To this end, a pseudo instruction called a *parallel copy*, is created to represent a set of copies that have to be executed in parallel. As explained further, the replacement of parallel copies by sequences of simple copies is handled later on. Algorithm 6 presents the corresponding pseudo-code that makes non-conventional SSA conventional. As already mentioned, SSA destruction of such form is straightforward. However Algorithm 6 can be slightly modified to directly destruct SSA by: removing line 15; replacing a'_i by a_0 in the following lines; adding "remove the ϕ-function" at line 19.

Algorithm 6: Critical Edge Splitting Algorithm for making non-conventional SSA form conventional.

```
1  begin
2      foreach $B$: basic block of the CFG do
3          let $(E_1, \dots, E_n)$ be the list of incoming edges of $B$;
4          foreach $E_i = (B_i, B)$ do
5              let $\mathrm{PC}_i$ be an empty parallel copy instruction;
6              if $B_i$ has several outgoing edges then
7                  create fresh empty basic block $B'_i$;
8                  replace edge $E_i$ by edges $B_i \to B'_i$ and $B'_i \to B$;
9                  insert $PC_i$ in $B'_i$;
10             else
11                 append $PC_i$ at the end of $B_i$;
12         foreach $\phi$-function at the entry of $B$ of the form $a_0 = \phi(B_1 : a_1, \dots, B_n : a_n)$ do
13             foreach $a_i$ (argument of the $\phi$-function corresponding to $B_i$) do
15                 let $a'_i$ be a freshly created variable ;
16                 add copy $a'_i \leftarrow a_i$ to $\mathrm{PC}_i$;
17                 replace $a_i$ by $a'_i$ in the $\phi$-function;
19
20 end
```

coalescer->coalescing several times in this paragraph

We stress that the above destruction technique has several drawbacks: first because of specific architectural constraints, region boundaries, or exception handling code, the compiler might not permit the splitting of a given edge; second, the resulting code contains many temporary-to-temporary copy operations. In theory, reducing the frequency of these copies is the role of the coalescing during the register allocation phase. A few memory- and time-consuming coalescing heuristics mentioned in Chapter **??** can handle the removal of these copies effectively. Coalescing can also, with less effort, be performed prior to the register allocation phase. As opposed to a (so-called conservative) coalescing during register allocation, this *aggressive* coalescing would not cope with the interference graph colorability. Further, the process of copy insertion itself might take a substantial amount of time and might not be suitable for dynamic compilation. The goal of Chapter **??** is to cope both with non-splittable edges and difficulties related to SSA destruction at machine code level, but also aggressive coalescing in the context of resource constrained compilation.

notation

Once ϕ-functions have been replaced by parallel copies, we need to sequentialize the parallel copies, i.e. replace them by a sequence of simple copies. This phase can be performed immediately after SSA destruction or later on, perhaps even after register allocation (see Chapter **??**). It might be useful to postpone the copy sequentialization since it introduces arbitrary interference between variables. As an example, $a_1 \leftarrow a_2 \parallel b_1 \leftarrow b_2$ (where $inst_1 \parallel inst_2$ represents two instructions $inst_1$ and $inst_2$ to be executed simultaneously) can be sequentialized into $a_1 \leftarrow a_2$; $b_1 \leftarrow b_2$ which would make b_2 interfere with a_1 while the other way round $b_1 \leftarrow b_2$; $a_1 \leftarrow a_2$ would make a_2 interfere with b_1 instead.

If we still decide to replace parallel copies into a sequence of simple copies immediately after SSA destruction, this can be done as shown in Algorithm 7. To see that this algorithm converges, one can visualize the parallel copy as a graph where nodes represent resources and edges represent transfer of values: the number of steps is exactly the number of cycles plus the number of non-self edges of this graph. The correctness comes from the invariance of the behavior of *seq*; *pcopy*. An optimized implementation of this algorithm will be presented in Chapter **??**.

Algorithm 7: Replacement of parallel copies with sequences of sequential copy operations.

```
begin
    let pcopy denote the parallel copy to be sequentialized;
    let seq = () denote the sequence of copies;
    while $\neg\,[\forall (b \leftarrow a) \in pcopy,\ a = b]$ do
        if $\exists (b \leftarrow a) \in pcopy$ s.t. $\nexists (c \leftarrow b) \in pcopy$ then
            /* b is not live-in of pcopy                          */
            append $b \leftarrow a$ to seq;
            remove copy $b \leftarrow a$ from pcopy;
        else
            /* pcopy is only made-up of cycles; Break one of them  */
            let $b \leftarrow a \in pcopy$ s.t. $a \neq b$;
            let $a'$ be a freshly created variable;
            append $a' \leftarrow a$ to seq;
            replace in pcopy $b \leftarrow a$ into $b \leftarrow a'$;
end
```

1.3 SSA Property Transformations

As discussed in Chapter **??**, SSA comes in different flavors. This section describes algorithms that transform arbitrary SSA code into the desired flavor. Making SSA *conventional* corresponds exactly to the first phase of SSA destruction (described in Section 1.2) that splits critical edges and introduces parallel copies (sequentialized later in bulk or on-demand) around ϕ-functions. As already discussed, this straightforward algorithm has several drawbacks addressed in Chapter **??**.

Making SSA *strict*, i.e. fulfill *the dominance property*, is as 'hard' as constructing SSA. Of course, a pre-pass through the graph can detect the offending variables that have definitions that do not dominate their uses. Then there are several possible single-variable ϕ-function insertion algorithms (see Chapter **??**) that can be used to patch up the SSA, by restricting attention to the set of non-conforming variables. The renaming phase can also be applied with the same filtering process. As the number of variables requiring repair might be a small proportion of all variables, a costly traversal of the whole program can be avoided by building the def-use chains (for non-conforming variables) during the detection pre-pass. Renaming can then be done on a per-variable basis or better (if pruned SSA is preferred) the reconstruction algorithm presented in Chapter **??** can be used for both ϕ-functions placement and renaming.

rephrased to summarized so that Algo can be in the same page.

The construction algorithm described above does not build *pruned* SSA form. If available, liveness information can be used to filter out the insertion of ϕ-functions wherever the variable is not live: the resulting SSA form is pruned. Alternatively, pruning SSA form is equivalent to a dead-code elimination pass after SSA construction. As use-def chains are implicitly provided by SSA form, dead-ϕ-function elimination simply relies on marking actual uses (non-ϕ-function ones) as *useful* and propagating *usefulness* backward through ϕ-functions. Algorithm 8 presents the relevant pseudo-code for this operation. Here *stack* is used to store useful and unprocessed variables, defined by ϕ-functions.

Put this in the text instead of in the Algo (gain space)

Algorithm 8: ϕ-function pruning algorithm

```
begin
    stack ← ();
    /* -- initial marking phase --                                    */
    foreach I : instruction of the CFG in dominance order do
        if I is $\phi$-function defining variable a then mark a as useless;
        else I is not $\phi$-function
            foreach x : source operand of I do
                if x is defined by $\phi$-function then
                    mark x as useful; stack.push(x);
    /* -- usefulness propagation phase --                             */
    while stack not empty do
        a ← stack.pop();
        let I be the $\phi$-function that defines a;
        foreach x : source operand of I do
            if x is marked as useless then
                mark x as useful; stack.push(x);
    /* -- final pruning phase --                                      */
    foreach I: $\phi$-function do
        if destination operand of I marked as useless then delete I;
end
```

To construct pruned SSA form via dead code elimination, it is generally much faster to first build *semi-pruned SSA form*, rather than minimal SSA form, and then apply dead code elimination. Semi-pruned SSA form is based on the observation that many variables are *local*, i.e. have a single live-range that is within a single basic block. Consequently, pruned SSA would not instantiate any ϕ-functions for these variables. Such variables can be identified by a linear traversal over each basic block of the CFG. All of these variables can be filtered out: minimal SSA form restricted to the remaining variables gives rise to the so called semi-pruned SSA form.

Pessimistic ϕ-function Insertion

Construction of minimal SSA, as outlined in Section 1.1, comes at the price of a sophisticated algorithm involving the computation of iterated dominance frontiers. Alternatively, ϕ-functions may be inserted in a *pessimistic* fashion, as detailed below. In the pessimistic approach, a ϕ-function is inserted at the start of each CFG node (basic block) for each variable that is live at the start of that node. A less sophisticated, or *crude*, strategy is to insert a ϕ-function for each variable at the start of each CFG node; the resulting SSA will not be pruned. When a pessimistic approach is used, many inserted ϕ-functions are redundant. Code transformations such as code motion, or other CFG modifications, can also introduce redundant ϕ-functions, i.e. make a minimal SSA program become non-minimal.

Restructured this paragraph. Mostly same sentences in a different order...

The application of *copy propagation* and rule-based ϕ-function rewriting, can removes many of these redundant ϕ-functions. As already mentioned in Chapter **??**, copy propagation can break the dominance property by propagating variable a_j through ϕ-functions of the form $a_i = \phi(a_{x_1}, \dots, a_{x_k})$ where all the source operands are syntactically equal to either a_j or $\perp$. If we want to avoid breaking the dominance property we simply have to avoid the application of copy propagation that involves $\perp$. A more interesting rule is the one that propagates a_j through a ϕ-function of the form $a_i = \phi(a_{x_1}, \dots, a_{x_k})$ where all the source operands are syntactically equal to either a_i or a_j. These ϕ-functions turn out to be 'identity' operations, where all the source operands become syntactically identical and equivalent to the destination operand. As such, they can be trivially elided from the program. Identity ϕ-functions can be simplified this way from inner to outer loops. To be efficient (otherwise as many iterations as the maximum loop depth might be necessary for copy propagation) variable def-use chains should be precomputed. When the CFG is reducible [2], this simplification produces minimal SSA. The underlying reason is that the def-use chain of a given ϕ-web is, in this context, isomorphic with a subgraph of the reducible CFG: all nodes except those that can be reached by two different paths (from actual non-ϕ-function definitions) can be simplified by iterative application of *T1* (removal of self edge) and

[2] A CFG is reducible if there are no jumps into the middle of loops from the outside, so the only entry to a loop is through its header. Section 1.4 gives a fuller discussion of reducibility, with pointers to further reading.

T2 (merge of a node with its unique predecessor) graph reductions. Figure 1.5 illustrates these graph transformations. To be precise, we can simplify an SSA program as follows:

1. Remove any ϕ-function $a_i = \phi(a_{x_1}, \dots, a_{x_k})$ where all $x_n \in \{i\}$. This corresponds to *T1* reduction.
2. Remove ϕ-function $a_i = \phi(a_{x_1}, \dots, a_{x_k})$ where all $x_n \in \{i, j\}$. Replace all occurrences of a_i by a_j. This corresponds to *T2* possibly followed by *T1* reduction.

added some edges

ai and aj were permuted

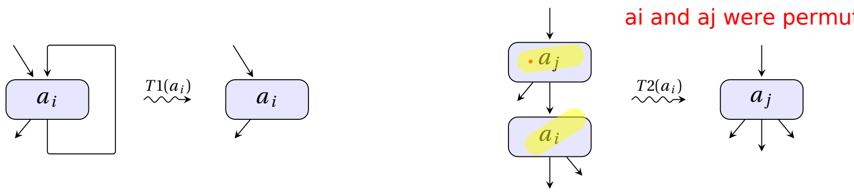


Fig. 1.5 *T1* and *T2* rewrite rules for graph reduction, applied to def-use relations between SSA variables

Used the term worklist from the beginning and then queue only for our algorithm.

This approach can be implemented using a worklist, which stores the candidate nodes for simplification. Using the graph made up of def-use chains (see Chapter **??**), the worklist can be initialized with successors of non-ϕ-functions. However for simplicity, we may initialize it with all ϕ-functions. Of course, if loop nesting forest information is available, the worklist can be avoided by traversing the CFG in a single pass from inner to outer loops, and in a topological order within each loop (header excluded). However since we believe the main motivation for this approach to be its simplicity, the pseudo-code shown in Algorithm 9 uses a work queue.

Was mentioning dependance graph which is not what we are usign. Summarized to have algo on the same page.

changed the caption

Algorithm 9: Removal of redundant ϕ-functions using rewriting rules and work queue

```
begin
    W ← () ;
    foreach I: $\phi$-function in program in reverse post-order do
        W.enqueue(I); mark I ;
    while W not empty do
        I ← W.dequeue(); unmark I;
        let I be of the form $a_i = \phi(a_{x_1}, \dots, a_{x_k})$;
        if all source operands of the $\phi$-function are $\in \{a_i, a_j\}$ then
            Remove I from program;
            foreach I′: instruction different from I that uses $a_i$ do
                replace $a_i$ by $a_j$ in I′;
                if I′ is a $\phi$-function and I′ is not marked then
                    mark I′; W.enqueue(I′);
end
```

This algorithm is guaranteed to terminate in a fixed number of steps. At every iteration of the while loop, it removes a ϕ-function from the work queue W. Whenever it adds new ϕ-functions to W, it removes a ϕ-function from the program. The number of ϕ-functions in the program is bounded so the number of insertions to W is bounded. The queue could be replaced by a worklist, and the insertions/removals done at random. The algorithm would be less efficient, but the end-result would be the same.

1.4 Additional reading

The early literature on SSA form [7, 8] introduces both phases of the construction algorithm we have outlined in this chapter, and discusses algorithmic complexity on common and worst-case inputs. These initial presentations trace the ancestry of SSA form back to early work on data flow representations by Shapiro and Saint [9].

Briggs et al [6] discuss pragmatic refinements to the original algorithms for SSA construction and destruction, with the aim of reducing execution time. They introduce the notion of semi-pruned form, show how to improve the efficiency of the stack-based renaming algorithm, and describe how copy propagation must be constrained to preserve correct code during SSA destruction.

There are numerous published descriptions of alternative algorithms for SSA construction, in particular for the ϕ-function insertion phase. The pessimistic approach that first inserts ϕ-functions at all control flow merge points and then removes unnecessary ones using simple *T1*/*T2* rewrite rules was proposed by Aycock and Horspool [2]. Brandis and Mössenböck [5] describe a simple, syntax-directed approach to SSA construction from well structured high-level source code. Throughout this textbook, we consider the more general case of SSA construction from arbitrary CFGs.

A *reducible* CFG is one that will collapse to a single node when it is transformed using repeated application of *T1*/*T2* rewrite rules. Aho et al [1] describe the concept of reducibility and trace its history in early compilers literature.

Sreedhar and Gao [10] pioneer linear-time complexity ϕ-function insertion algorithms based on DJ-graphs. These approaches have been refined by other researchers. Chapter **??** explores these alternative construction algorithms in depth.

Blech et al [3] formalize the semantics of SSA, in order to verify the correctness of SSA destruction algorithms. Boissinot et al [4] review the history of SSA destruction approaches, and highlight misunderstandings that led to incorrect destruction algorithms. Chapter **??** presents more details on alternative approaches to SSA destruction.

There are instructive dualisms between concepts in SSA form and functional programs, including construction, dominance and copy propagation. Chapter **??** explores these issues in more detail.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles Techniques and Tools*. Addison Wesley, 1986.
2. John Aycock and Nigel Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000.
3. Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. Optimizing code generation from ssa form: A comparison between two formal correctness proofs in isabelle/hol. *Electronic Notes in Theoretical Computer Science*, 141(2):33–51, 2005.
4. Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–125, 2009.
5. Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, Nov 1994.
6. P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–882, 1998.
7. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM.
8. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
9. Robert M. Shapiro and Harry Saint. The representation of algorithms. Technical Report RADC-TR-69-313, Rome Air Development Center, September 1969.
10. Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, 1995.