CHAPTER 2

Properties and flavours

P. Brisk

Progress: 90%

Text and figures formating in progress

2.1 Preliminaries

Recall from the previous chapter that a procedure is in SSA form if every variable is defined once, and every use of a variable corresponds to exactly one definition. Many variations, or flavors, of SSA form that satisfy these criteria can be defined, each offering its own considerations. For example, different flavors vary in terms of the number of ϕ-functions, which affects the size of the intermediate representation; some are more difficult to construct, maintain, and destruct compared to others. This chapter explores these SSA flavors and provides insight onto the contexts that favor some over others.

2.2 Def-Use and Use-Def Chains

Under SSA form, each variable is defined once. Def-use chains are data structures that provide for the single definition of a variable the set of all its uses. Each use-def chain inversely provides for each use of a variable its unique definition. As we will illustrate further in the book (see Chapter **??**) def-use chains are useful for forward data-flow analysis as they provide direct connections that shorten the propagation distance between nodes that generate and use data-flow information.

Because of its single definition per variable property, SSA form simplifies def-use and use-def chains in several ways. First, SSA form simplifies def-use chains as it combines the information as early as possible. This is illustrated by Figure 2.1 where the def-use chains in the non-SSA program requires as many merge as x is used while the corresponding SSA form allows early and more efficient combination.

Second, as it is easy to associate to each variable its single defining operation, use-def chains can be explicitly represented and maintained almost for free. As this constitutes the skeleton of the so called SSA graph (see Chapter 18), when considering a program under SSA form, use-def chains are implicitly considered as a given. The explicit representation of use-def chains simplifies backward propagation, which favors algorithms such as dead code elimination.

For forward propagation, def-use chains being just the reverse of use-def chains, computing it is also easy, and maintaining it can be done without much efforts either. However, even without def-use chains, some lightweight forward propagation algorithms such as copy-folding are possible and show to be already quite efficient using only use-def chains: if loops are conservatively ignored, operations can be processed in topological order so that many definitions are processed prior to the uses.

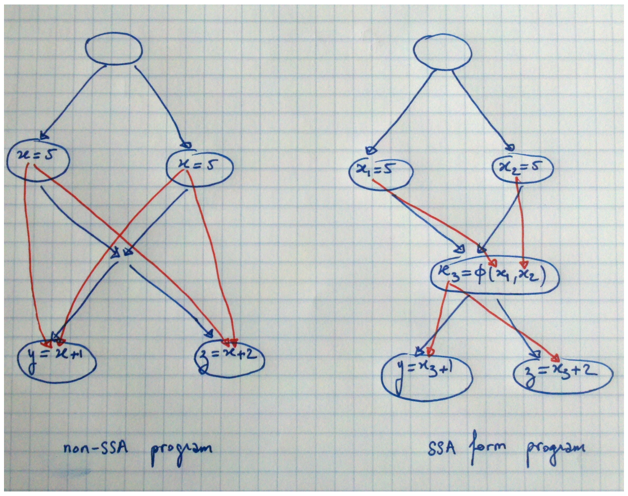


Fig. 2.1 Def-use chains (in red) for non-SSA form and its corresponding SSA form program

2.3 Minimality

SSA construction is a two-phase process: placement of ϕ-functions, followed by renaming. The goal of the first phase is to generate a code that fulfills the further defined single reaching-definition property. Minimality is an additional property of the code with ϕ-functions inserted, but prior to renaming; Chapter **??** describes the classical SSA construction algorithm in detail, while this section focuses primarily on describing the minimality property.

A definition D of variable v *reaches* a point p in the CFG if there exists a path from D to p that does not pass through another definition of v. We say that a code has the *single reaching-definition property* iff no program point can be reached by two definitions of the same variable. Under the assumption that the single reaching-definition property is fulfilled, the *minimality property* states the minimality of the number of inserted ϕ-functions.

This property can be characterized using the notion of join sets that we introduce next. Let n_1 and n_2 be distinct basic blocks in a CFG. A basic block n_3, which may or may not be distinct from n_1 or n_2, is a *join node* of n_1 and n_2 if there exist at least two non-empty paths, i.e., paths containing at least one CFG edge, from n_1 to n_3 and from n_2 to n_3, respectively, such that n_3 is the only basic block that occurs on both of the paths. In other words, the two paths converge at n_3 and no other CFG node. Given a set S of basic blocks, n_3 is a join node of S if it is the join node of at least two basic blocks in S. The set of join nodes of set S is denoted by the set $\mathcal{J}(S)$.

Intuitively, a join set corresponds to the placement of ϕ-functions. In other words, if n_1 and n_2 are basic blocks that both contain the definition of a variable v, then we ought to instantiate ϕ-functions for v at every basic block in $\mathcal{J}(n_1, n_2)$. Generalizing this statement, if D_v is the set of basic blocks containing definitions of v, then ϕ-functions should be instantiated in every basic block in $\mathcal{J}(D_v)$. As inserted ϕ-functions are themselves definition points, some new ϕ-functions should be inserted at $\mathcal{J}(D_v \cup \mathcal{J}(D_v))$. Actually it turns out that $\mathcal{J}(S \cup \mathcal{J}(S)) = \mathcal{J}(S)$ so the join set of the set of definition points of a variable characterizes exactly the minimum set of program points where ϕ-functions should be inserted.

We are not aware of any optimizations that require a strict enforcement of minimality property. However, placing ϕ-functions only at the join sets can be done easily using a simple topological traversal of the CFG as described in Section 4.5. Classical techniques place ϕ-functions of a variable v at $\mathcal{J}(D_v, r)$, with r the entry node of the CFG. There are good reasons for that as we will explain further. Finally, as explained in Section 3.3 for reducible flow graphs, some copy-propagation engine can easily turn a non-minimal SSA code into a minimal one.

2.4 Strict SSA Form. Dominance Property

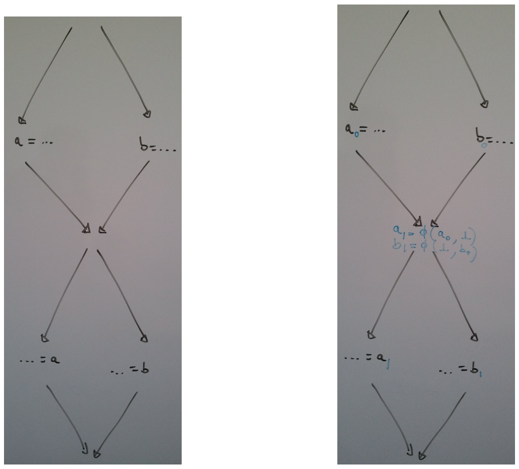
A procedure is defined to be *strict* if every variable is defined before it is used along every path from the entry to exit point; otherwise, it is *non-strict*. Some languages, such as Java, impose strictness as part of the language definition; others, such as C/C++, impose no such restrictions. The code in Figure 2.2(a) is non-strict as there exists a path from the entry to the use of a that does not go through the definition. If this path is taken through the CFG during the execution, then a will be used without ever being assigned a value. Although this may be permissible in some cases, it is usually indicative of a programmer error or poor software design.

Under SSA, because there is only a single (static) definition per variable, strictness is equivalent to the *dominance property*: each use of a variable is dominated by its definition. In a CFG, basic block n_1 *dominates* basic block n_2 if every path in the CFG from the entry point to n_2 includes n_1. By convention, every basic block in a CFG dominates itself. Basic block n_1 *strictly dominates* n_2 if n_1 dominates n_2 and $n_1 \neq n_2$. We use the symbols $n_1 \mathit{dom}\, n_2$ and $n_1 \mathit{sdom}\, n_2$ to denote dominance and strict dominance respectively.

Adding a (undefined) pseudo-definition of each variable to the procedure's entry point ensures strictness. The single reaching-definition property discussed previously mandates that each program point be reachable by exactly one definition (or pseudo-definition) of each variable. If a program point U is a use of variable v, then the reaching definition D of v will dominate U; otherwise, there would be a path from the CFG entry node to U that does not include D. If such a path existed, then the program would not be in SSA form, and a ϕ-function would need to be inserted somewhere in $\mathcal{J}(r, D)$ as in our example of Figure 2.2(b) where $\perp$ represents the undefined pseudo-definition. The so called *minimal SSA form* is a variant of SSA form that satisfies both the minimality and dominance properties. As shall be seen in Chapter **??**, minimal SSA form is obtained by placing the ϕ-functions of variable v at $\mathcal{J}(D_v, r)$ using the formalism of dominance frontier. If the original procedure is non-strict, conversion to minimal SSA will create a strict SSA-based representation. Here, strictness refers solely to the SSA representation; if the input program is non-strict, conversion to and from strict SSA form cannot address errors due to uninitialized variables. To finish with, the use of an implicit pseudo-definition in the CFG entry node to enforce strictness does not change the semantics of the program by no means.

SSA with dominance property is useful for many reasons that directly originate from the structural properties of the variable live-ranges. The immediate dominator or idom of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n. All nodes but the entry node have immediate dominators. A dominator tree is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree with the entry node as root. Each live-range is a sub-tree of the dominator tree. Among other consequences of this property, we can cite the ability to design a fast and efficient method to query whether a variable is live at point q or an iteration free algorithm to computes liveness sets (see Chapter **??**); this property allows also efficient algorithms to test whether two variables interfere (see Chapter **??**).

Another elegant consequence is that the interference graph belongs to a special class of graphs called chordal graphs, which are the intersection graphs of a set of sub-trees of a tree. Chordal graphs are significant because several problems that are NP-complete on general graphs have efficient linear-time solutions on chordal graphs, including graph coloring, which plays an important role in register allocation in compilers. In particular, a traversal of the dominator tree called a tree-scan



(a) Non-strict code (b) SSA with dominance property

Fig. 2.2 A non-strict code and its corresponding strict SSA form. The presence of $\perp$ indicates a use of an undefined value.

can color all of the variables in the program, without requiring the explicit construction of an interference graph. The trees-scan algorithm can be used for register allocation, which is discussed in greater detail in Chapter **??**.

As we have already mentioned, most ϕ-function placement algorithms are based on the notion of dominance frontier (see chapters **??** and **??**) consequently do provide the dominance property. As we will see in Chapter **??**, this property can be broken by copy-propagation: in our example of Figure 2.2(b), the argument a_1 of the copy represented by $a_2 = \phi(a_1, \perp)$ can be propagated and every occurrence of a_2 can be safely replaced by a_1; the now identity ϕ-function can then be removed obtaining the initial SSA but non strict code. Making a non-strict SSA code, strict is somehow as "difficult" as SSA construction (actually we need a pruned version as described below). Still the "strictification" usually concerns only a few variables and a restricted region of the CFG: the incremental update described in Chapter **??** will do the work with less efforts.

2.5 Pruned SSA Form

One drawback of Minimal SSA form is that it may place ϕ-functions for a variable at a point in the control flow graph where the variable was not actually live prior to SSA. Many program analyses and optimizations, including register allocation, are only concerned with the region of a program where a given variable is live. The primary advantage of eliminating those dead ϕ-functions over minimal SSA form is that it has far fewer ϕ-functions in most cases. It is possible to construct such a form while still maintaining the minimality and dominance properties otherwise. The new constraint is that every *use point* for a given variable must be reached by exactly one definition, as opposed to all program points. Pruned SSA form satisfies these properties.

Under minimal SSA, ϕ-functions for variable v are placed at the entry points of basic blocks belonging to the set $\mathcal{J}(S,r)$. Under pruned SSA, we suppress the instantiation of a ϕ-function at the beginning of a basic block if v is not live at the entry point of that block. One possible way to do this is to perform liveness analysis prior to SSA construction, and then use the liveness information to suppress the placement of ϕ-functions as described above; another approach is to construct minimal SSA and then remove the dead ϕ-functions using dead code elimination. Details can b nd in Chapter 3.

Figure 2.3(a) shows an example of minimal non pruned SSA. The corresponding pruned SSA form would remove the dead ϕ-function that defines Y_3.

```
if P1                                  if P1
   then do                                then do
      Y1  ←  1                               Y1  ←  1
      use of Y1                              use of Y1
   end                                    end
   else do                                else do
      Y2  ←  X1                              Y2  ←  X1
      use of Y2                              use of Y2
   end                                    end
Y3 ←  φ(Y1,Y2)                         Y3 ←  φ(Y1,Y2)
   ...                                    ...
if P1
   then  Z1  ←  1
   else  Z2  ←  X1
Z3 ←  φ(Z1,Z2)
use of Z3                              use of Y3
```

Fig. 2.3 Non pruned SSA form allows value numbering to determine that Y_3 and Z_3 have the same value.

2.6 Conventional and Transformed SSA Form

In many non-SSA and graph coloring based register allocation schemes, register assignment is done at the granularity of webs. In this context, a web is the maximum unions of def-use chains that have either a use or a def in common. As an example, the code of Figure a) leads to two separate webs for variable a. The conversion to minimal SSA form replaces each web of a variable v in the pre-SSA program with some variables version v_i. In pruned SSA, these variables version partition the live-range of the web: at every point in the procedure where the web is live, *exactly* one variable v_i is also live; and none of the v_i are live at any point where the web is not.

Based on this observation, we can partition the variables in a program that has been converted to SSA form into ϕ-equivalence classes that we will refer as ϕ-webs. We say that x and y are *ϕ-related* to one another if they are referenced by the same ϕ-function, i.e., if x and y are either parameters or defined by the ϕ-function. The transitive closure of this relation defines an equivalence relation that partitions the variables defined locally in the procedure into equivalence classes, the ϕ-webs. Intuitively, the ϕ-equivalence class of a resource represents a set of resources "connected" via ϕ-functions. For any freshly constructed SSA code, the ϕ-webs exactly correspond to the register web of the original non-SSA code.

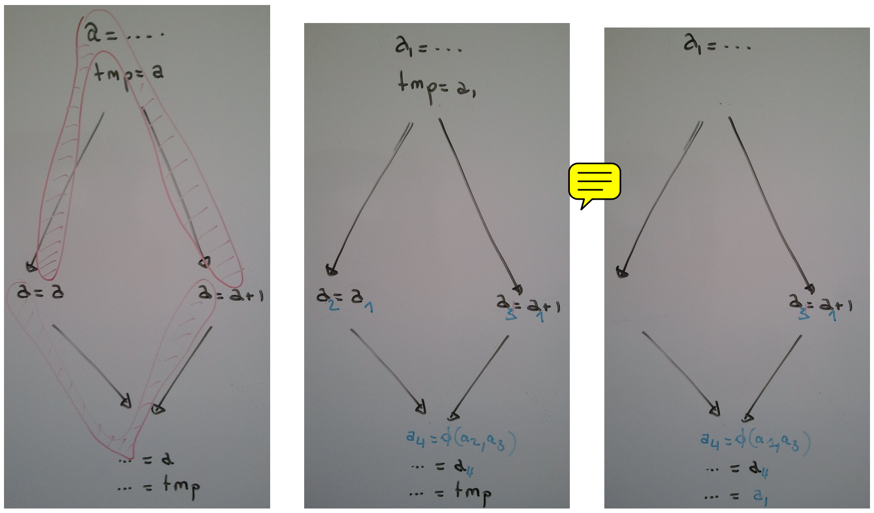
Conventional SSA form (C-SSA) is defined as SSA form for which each ϕ-web is interference-free. Many program optimizations such as copy-propagation may transform a procedure from conventional to a non-conventional (T-SSA for Transformed-SSA) form, in which some variables belonging to the same ϕ-web interfere with one another. Figure 2.4(c) shows the corresponding transformed SSA form of our previous example: here variable a_1 interfere with variables a_2, a_3, and a_4.

Bringing back the conventional property of a T-SSA code is as "difficult" as destructing SSA (see Chapter **??**). The translation out of conventional SSA form is straightforward: each ϕ-web can be replaced with a single variable; all definitions and uses are renamed to use the new variable, and all ϕ-functions involving this equivalence class are removed. SSA destruction starting from non-conventional SSA form can be performed through a conversion to conventional SSA form as an intermediate step. This conversion is achieved by inserting copy operations that dissociate interfering variables from the connecting ϕ-functions. As those copy instructions will have to be inserted at some points to get rid of ϕ-functions, for machine level transformations such as register allocation or scheduling, T-SSA provides an inaccurate view of the resource usage. Another motivation for sticking on C-SSA is that the names used in the original program might help capturing some properties otherwise difficult to discover. Lexical partial redundancy elimination (PRE) as described in Chapter **??** illustrates this point.

Apart from those specific examples most current compilers choose not to maintain the conventional property. Still, we should outline that, as later described in Chapter **??**, checking if a given ϕ-web is (and if necessary turning it back to) interference-free can be done in linear time (instead of the naive quadratic time algorithm) in the size of the ϕ-web.

2.7 A Stronger Definition of Interference

Throughout this chapter, two variables have been said to interfere if their live ranges intersect. Intuitively, two variables with overlapping lifetimes will require two distinct storage locations; otherwise, a write to one variable will overwrite the value of the other. In particular, this definition has applied to the discussion of interference graphs and the definition of conventional SSA form, as described above.



(a) the two webs of variable a

(b) the two corresponding ϕ-webs $\{a_1\}$ and $\{a_2, a_3, a_4\}$

(c) after propagating a_1, SSA form is not conventional anymore

Fig. 2.4 Non-SSA register webs and corresponding SSA ϕ-webs; conventional and corresponding transformed SSA form

Although it suffices for correctness, this is a fairly restrictive definition of interference, based on static considerations. The ultimate notion of interference, that is obviously undecidable because of a reduction to the Halting problem, should decide for two distinct variables whether there exists an execution for which they simultaneously hold two different values. Several “static” extensions to our simple definition are still possible, in which, under very specific conditions, variables whose live ranges overlap one anothe[illegible]y not interfere. We present two examples.

Firstly, consider the double-diamond graph of Figure 2.2(a) again, which although non-strict, is correct as soon as the two if-conditions are the same. Even if a and b are unique variables with overlapping live ranges, the paths along which a and b are respectively used and defined are mutually exclusive with one another. In this case, the program will either pass through the definition of a and the use of a, or the definition of b and the use of b, since all statements involved are controlled by the same condition, albeit at different conditional statements in the program. Since only one of the two paths will ever execute, it suffices to allocate a single storage location that can be used for a or b. Thus, a and b do not actually interfere with one another. A simple way to refine the interference test is to check if one of the variable is live at the definition point of the other. This relaxed but correct notion of interference would not make a and b of Figure 2.2(a) interfere while variables a_1 and b_1 of Figure 2.2(b) would still interfere. This example illustrates the fact that live-range splitting required here to make the code fulfill [illegible] dominance property may lead to less accurate analysis results. As far as the interfere is concerned, for a SSA code with dominance property, the two notions are strictly equivalent: two live-ranges intersect iff one contains the definition of the other.

Secondly, consider two variables u and v, whose live ranges overlap. If we can prove that u and v will always hold the same value at every place where both are live, then they do not actually interfere with one another. Since they always have the same value, a single storage location can be allocated for both variables, because there is only one unique value between them. Of course, this new criteria is in general undecidable. Still, a technique such as value numbering that is straightforward to implement under SSA (See 8) can make a fairly good job, especially in the presence of a code with many variable-to-variable copies, such as one obtained after a naive SSA destruction pass (See Chapter **??**). In that case (See Chapter **??**), the difference between the refined notion of interference and non-value based one is significant.

This refined notion of interference has significant implications if applied to SSA form. In particular, the interference graph of a procedure is no longer chordal, as any edge between two variables whose lifetimes overlap could be eliminated by this property.

2.8 Further readings

The advantages of def-use and use-def chains provided for almost free under SSA are well illustrated in chapters 8 and **??**.

The notion of minimal SSA and a corresponding efficient algorithm to compute it were introduced by Cytron et al. in [**?**]. For this purpose they extensively develop the notion of dominance frontier of a node n, $\mathcal{DF}(n) = \mathcal{J}(n, r)$. The fact that $\mathcal{J}^+(S) = \mathcal{J}(S)$ have been actually discovered later, and Wolfe gives a simple proof of it in [**?**]. More details about the theory on (iterated) dominance frontier can be found in chapters **??** and **??**. The post-dominance frontier, which is its symmetric notion, also known as the control dependence graph finds many applications. Further discussions on control dependence graph can be found in Chapter **??**.

Most SSA papers implicitly consider the SSA form to fulfill the dominance property. The first technique that really exploits the structural properties of the strictness is the fast SSA destruction algorithm developed by Budimlic et al. in [**?**] and revisited in Chapter **??**.

The notion of pruned SSA have been introduced in [**?**]. The example of Figure 2.3 to illustrate the difference between pruned and non pruned SSA have been borrowed from [**?**]. The notions of conventional and transformed SSA were introduced by Sreedhar et al. in their seminal paper [**?**] for destructing SSA form. The description of the existing techniques to turn a general SSA into either a minimal, a pruned, a conventional, or a strict SSA is provided in Chapter **??**.

The ultimate notion of interference was first discussed by Chaitin in his seminal paper [**?**] that presents the graph coloring approach for register allocation. His interference test is similar to the refined test presented in this chapter. In the context of SSA destruction, Chapter **??**, addresses the issue of taking advantage of the dominance property with this refined notion of interference.