Chapter 1

Properties and flavours —(*P. Brisk*)

Writing: 80%

1 Preliminaries

definition of SSA provided in previous chapter. useful to recall only if this is to stress what the definition _do not_ enforce.

Any operation of the form $v \leftarrow \dots$ that assigns a value to variable v is called a *definition* of v. Any operation of the form $\dots \leftarrow v$ that reads a value of v is called a *use* of v. One of the key properties of SSA Form is that each variable is defined only once; however, renaming each variable alone is insufficient, and ϕ-functions are needed to merge the disparate redefinitions of each variable. The remaining sections of this chapter provide the theoretical foundations that describe precisely how ϕ-functions are inserted, and then derives important properties of the SSA Form that result.

phi-functions

2 Dominance

we do not care about post-dominance here.

The concepts of *dominance*, introduced in this section, and *post-dominance*, introduced in the following section, form the foundations of a collection of compiler techniques called *control flow analysis*. Intuitively dominance and post-dominance respectively tell us which basic blocks are *guaranteed* to execute before or after others. Historically, this information has been used to identify hierarchies of nested loops. Although this problem may seem trivial, given the prevalence of programming constructs to express loops, such as *for*, *while* and *do-while*, the problem becomes significantly more complicated when trying to account for *irreducible loops*, which can be constructed using arbitrary *goto* statements [?]. Our interest, however, is SSA Form rather than loop analysis; as we shall see, extensions to these basic definitions, in the sections that follow, are necessary to understand the criteria for instantiating ϕ-functions during the construction of SSA Form.

dominance

useless detour

to be moved in the appendix. You can put the defintion of dominance in one sentence just to recall and refer to the appendix.

~~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 $n_1 dom n_2$ and $n_1 sdom n_2$ to denote dominance and strict dominance respectively.~~

~~The *immediate dominator* of a basic block n_2, denoted by $idom(n_2)$, is the basic block n_1 such that $n_1 sdom n_2$, and there does not exist any basic block n_3 such that $n_1 sdom n_3 sdom n_2$. Intuitively, this means that n_1 is the closest basic block to n_2 that strictly dominates n_2. Every basic block other than the entry point has exactly one immediate dominator.~~

appendix

~~The *dominator tree* is a data structure that links every basic block to its immediate dominator. The entry point of the CFG, itself a basic block, is the root of the dominator tree. If N is the set of basic blocks in the CFG and r represents the entry point, then the dominator tree is the graph $T = (N, E_T)$, where $E_T = \{(n, idom(n)) | n \in N \wedge n \neq r\}$. In other words, each edge in the dominator tree points from each basic block, other than r, to its immediate dominator.~~

3 Post-dominance

appendix

~~Post-dominance is similar in principle to dominance, as we will see below. In fact, all of the post-dominance information described below can be constructed using the basic methods required to compute dominance information. It suffices to reverse every edge in the CFG, and then swap the roles of the entry and exit node. The dominance information computed for the resulting CFG is wholly equivalent to the post-dominance information for the original CFG.~~

~~In a CFG, basic block n_2 *post-dominates* basic block n_1 if every path in the CFG from n_1 to the exit point includes n_2. By convention, every basic block in the CFG post-dominates itself. Basic block n_2 *strictly post-dominates* n_1 if n_2 post-dominates n_1 and $n_2 \neq n_1$. We use the symbols $n_2 pdom n_1$ and $n_2 spdom n_1$ to denote post-dominance and strict post-dominance respectively.~~

~~The *immediate post-dominator* of a basic block n_1, denoted by $ipdom(n_1)$, is the basic block n_2 such that $n_2 spdom n_1$, and there does not exist any basic block n_3 such that $n_2 spdom n_3 spdom n_1$. Intuitively, this means that n_2 is the closest basic block to n_1 that strictly post-dominates n_1. Every basic block other than the exit point has exactly one immediate post-dominator.~~

~~The *post-dominator tree* is a data structure that links every basic block to its immediate post-dominator. The exit point of the CFG is the root of the post-dominator tree. If N is the set of basic blocks in the CFG, and t is the unique exit point, then the post-dominator tree is the graph $P = (N, E_P)$, where $E_P = \{(n, ipdom(n)), n \in N \wedge n \neq t\}$. In other words, each edge in the post-dominator tree points from each basic block, other than t, to its immediate post-dominator.~~

4 Join Sets and Dominance Frontiers

appendix

join node

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 $\mathcal{J}(S)$.

join-set & phi-function placement

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 S is the set of basic blocks containing definitions of v, then ϕ-functions should be instantiated in every basic block block in $\mathcal{J}(S)$.

redundant with construction chapter

That being said, we still require an efficient method to compute join sets. In principle, we may require join sets for every subset of basic blocks in the CFG, although in principle, we are likely to require fewer. For this reason, we will turn, briefly, to an alternative structure based on dominance, rather than paths in a graph.

DF is more efficient than join sets

The *dominance frontier* of a basic block n_1 in a CFG, denoted $DF(n_1)$, is the set of basic blocks X, such that n_1 *does not* strictly dominate X, but *does* dominate at least one predecessor block of X. The dominance frontier of a set S of basic blocks, denoted $DF(S)$, is the union of the dominance frontiers of the elements of S.

The following section will make a connection between join sets and dominance frontiers.

5 Iterated Join Sets and Dominance Frontiers

Let S be a set of basic blocks, and let $\mathcal{J}^0(S) = \mathcal{J}(S)$. Now, let us recursively define $\mathcal{J}^i(S) = \mathcal{J}(S \cup \mathcal{J}^{i\backslash 1}(S))$. By construction, it is straightforward to see that $\mathcal{J}^i(S) \supseteq \mathcal{J}^{i-1}(S)$ for all i, as there is no process to facilitate vertex removal. If we compute these sets iteratively, we will eventually converge to a situation where $\mathcal{J}^i(S) = \mathcal{J}^{i-1}(S)$ in a finite number of steps, as each CFG contains a finite number of basic blocks. The resulting set, denoted by $\mathcal{J}^+(S)$, is called the *iterated join set* of S.

iterated join set

We can do something similar with dominance frontiers. Given a set S of basic blocks, let $DF^0(S) = DF(S)$, and let $DF^i(S) = DF(S \cup DF^{i-1}(S))$. Once again, we can iteratively compute $DF^i(S)$ from $DF^{i-1}(S)$, stopping when the two sets are equal. The resulting set, denoted by $DF^+(S)$ is called the *iterated dominance frontier* of S.

iterated DF

There is, in fact, a connection between iterated join sets and iterated dominance frontiers. Let X be any set of CFG nodes that contains the CFG entry node. Cytron et al. [6] proved that $\mathcal{J}^+(X) = DF^+(X)$. Weiss [8] proved a slightly stronger version, that $\mathcal{J}(X) = DF^+(X)$. Weiss also conjectured that $\mathcal{J}^+(S) = \mathcal{J}(S)$ for any set of CFG nodes S, and this conjecture was formally proven by Wolfe [9].

J(X+r)=DF+(X+r)

For reasons that we will discuss in greater detail below, each variable in an SSA Form program has an *implicit* definition at the CFG entry point. Based on the results above, either the join set or iterated dominance frontier could be used, in principle to instantiate ϕ-functions. Cytron et al. [6] compute iterated dominance frontiers using a worklist algorithm. As noted by Wolfe [9], join and iterated join sets are too complicated to compute directly, and no algorithms exist that call for their explicit use, outside of any context where iterated dominance frontiers suffice.

DF+ for phi placement

Intuitively, dominance frontiers are not enough, which is why iterated dominance frontiers are needed. Ignoring the subtle differences between dominance frontiers and join sets, consider a variable v defined in basic blocks n_1 and n_2. Initially, we will instantiate a $\phi-$function for v in some basic block $n_3 \in DF(n_1, n_2)$; however, before renaming, this ϕ-function itself is a new definition of v, so we will need to instantiate additional ϕ-functions in $DF(n_3)$ as well. Taken ad-infinitum, this line of reasoning is precisely why we use iterated dominance frontiers instead of dominance frontiers.

6 Split Sets and (Iterated) Post-dominance Frontiers

appendix

We have already noted the existence of a duality between the dominance and post-dominance relations. The *dual* of a join set is called a *split set*: given two distinct basic blocks n_2 and n_3, basic block n_1 belongs to their split set, denoted $\S(n_1, n_2)$ if there are non-empty paths from n_3 to n_1 and n_2 respectively, such that the only basic block common to both paths is n_3. Clearly, if all of the CFG edges are reversed, n_3 would belong to the join set of n_1 and n_2. Based on Wolfe's [9] proof, there is no need to define an *iterated split set*, as it is equivalent to the split set.

Similarly, the dominance fronter can be generalized to the *post-dominance frontier* by replacing the dominance with the post dominance relation in the definition. For a set of basic blocks, S, we let $PDF(S)$ denote the post-dominance frontier of S. Furthermore, we can define an *iterated post-dominance frontier* in a manner that is exactly similar to the iterated dominance frontier. The iterated post-dominance frontier of a set S of vertices is denoted $PDF^+(S)$.

(iterated) split sets
(iterated) DF

7 Philip's comment

????

From the outline, I budgeted 2 pages for the text of the preceding sections, and 2 pages for illustrations.

?

8 SSA Form, Def-Use, and Use-Def Chains

redundant with construction

A procedure is defined to be in SSA Form if it satisfies two properties. Firstly, each variable can only be defined once. This property is easily achieved: assuming that varible v is defined k times in the procedure, rename each of the definites to $v_1, v_2, \cdots, v_k$ respectively. The second property is that each use of v must now be renamed to correspond to precisely one definition. Without loss of generality, if there is a use of v in a basic block that belongs to the join set of two (or more) definitions v_i and v_j, then the second property is implicitly unsatisfied: renaming this use of v to be a use of v_i or v_j will not maintain correct program semantics. In order to rectify the situation, a ϕ-function must be instantiated to merge the definitions of v_i and v_j into a new variable v_l, and the use of v is replaced with a use of v_l.

SSA form: one definition
+ phis to fix pb

Note: insert a figure here to illustrate.

refer to analysis that use DU and/or UD chains

Two important data structures that are used in compilers are *Definition-Use (DU) Chains* and *Use-Definition (UD) Chains*. A definition D of variable v reaches a use U of v if there is a path in the CFG from D to U that does not include another definition of v. A DU Chain connects D to all uses of v that are reachable from D; a UD Chain connects a use U of v to all of the definition of v from which U is reachable.

Under SSA Form, as mentioned above, each variable is defined once and each use corresponds to a single definition. Therefore, there is exactly one DU Chain per variable (the definition points to all of the uses), and one UD Chain per use, which points to the single definition. Thus, the UD chains are explicit, and do not need to be represented.

UD chains are explicit
under SSA

Note: Insert a figure here to illustrate what happens to DU and UD chains by the conversion to SSA Form.

9 Dead Code Elimination Under SSA

simply say that UD chains can be used to perform DCE using a backward propagation from really useful instructions as described in the propagation engine chapter.

Note: In an email on October 19, 2009, Fabrice suggested that I include this section. If I choose to write it and include it here, I will base the discussion on The SSA DCE algorithm from Robert Morgan's textbook, which was used in MachSUIF (among other places). Alternativly, I could use the DCE algorithm from the classic Cytron et al. 1991 paper.

I have not yet written this section, because I am not sure whether an introductory chapter is the appropriate place to insert a discussion of DCE, or whether it is better moved to a chapter on SSA-based optimizations.

Dead code elimination under SSA

10 SSA with 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 [2]; 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.

strict code

It is possible to define SSA Form in a manner that ensures strictness, even if the original procedure itself is non-strict; it is also possible to define SSA Form in a way that does not ensure strictness. The designer of an SSA-based compiler must determine which definition is most appropriate.

SSA definition imposes nothing about strictness

give an example

introduce v & v_0 here

Any existing SSA construction algorithm can ensure strictness: it suffices to add a pseudo-definition of each variable at the entry point of the procedure. This pseudo-definition does not affect the live range of the variable: it is only used to guide the placement of ϕ-functions. Any SSA construction algorithm that uses iterated dominance frontiers, as we will shortly show, uses this implicit definition. refer to construction chapter

just add a pseudo-def @ r: vo

~~Let v_0 denote the pseudo-definition corresponding to variable v in a pre-SSA procedure~~. Any instruction that uses v_0 indicates a likely programmer error. The instruction will use a non-initialized variable; in the general case, this leads to unpredictable program behavior whenever this instruction is executed. Similarly, a ϕ-function may also contain v_0 as a parameter. If the corresponding path is taken into the basic block containing the ϕ-function, then an uninitialized variable (denoted by the pseudo- definition) will be copied to the variable defined by the ϕ-function. As these programs are still legal, depending on the language definition, the best that a compiler can do is to present a warning to the user in order to alert him or her of the situation.

how to handle vo

redundant

Let S_v be the set of basic blocks containing definitions of variable v, and let n_0 denote the entry point of the CFG. Conceptually, non-strict SSA Form can be constructed by placing ϕ-functions at the entry points of each basic block belonging to $J(S_v) = J^+(S_v)$, i.e., using join sets. Using iterated dominance frontiers, in contrast, ensures strict SSA Form, since $DF^+(S_v) = J(S_v \cup n_0)$.

J => non-strict
DF+ => strict

As there are no known efficient algorithms to compute join sets, the most straightforward way to compute non-strict SSA Form is to first built strict SSA Form and then remove ϕ-functions that are otherwise extraneous. Consider, for example, a ϕ-function that defines variable v_j: one of the parameters is another definition of v, denoted by v_i, while *all* of the remaining parameters are v_0: the implicit definition. It is possible, in this

strict + copy-prop
=> non-strict

copy-propagation (that you can describe briefly by refering to the propagation engine chapter in a few sentences) can make a strict SSA non-strict. BUT, it will not lead to minimal non-strict SSA form (because of circuits). Making a non-strict strict can be done using the maintain algorithm described in Sebastian's chapter.

talk about strict until here

case, to eliminate this ϕ-function by replacing all uses of v_j with uses of v_i instead. Although this destroys the dominance property, it does reduce the number of ϕ-functions in the program.

Note: Insert an example here, as described in an email from Fabrice: 'example of a double diamond with a def and used on the left and b def and used on the right: no interference'. I also have some decent examples from a paper published at IWLS 2007 (no official proceedings).

There are certainly situations where one would want the dominance property as well. Most of these situations exploit properties relating to liveness and interference, which are briefly outlined in the following sections of this chapter, and are described in much greater detail in subsequence chapters.

non-strict => strict
dominance frontier may
be useful

spilling is no more SSA

~~It is also important to note that many program transformations may cause strict SSA Form to become non-strict. Examples of such transformations include copy propagation and spilling. Techniques to convert non-strict to strict SSA Form without explicitly destroying and rebuilding SSA Form will be discussed in Section ...~~

~~(requires coordination with Sebastian). (possibly put an example here showing how copy propagation destroys strictness property; spilling works too).~~

redundant

11 Minimal SSA Form

instead of explaining how
we compute
minimal SSA, provide the
definition instead.
Still you can say that
using DF we can build
minimal SSA (and refer
to the construction chap).

Minimal SSA Form is computed in a straightforward manner, given the preceding discussion. Let S_v be the set of basic blocks containing definitions of variable v. Then ϕ-functions for v are instantiated in every basic block in $DF^+(S_v)$. This is followed by a pass over the program that renames each variable so that it has a unique definition, and renames each use to correspond precisely to one definition. Detailed pseudocode for these procedures can be found in ...

minimal SSA construction

it depends on your client. For
PRE for example you may
need the information further
than the last use and minimal
SSA _is_ minimal.

As we will soon see, Minimal SSA Form is hardly minimal, as other algorithms that insert considerably fewer ϕ-functions have been developed; however, Minimal SSA Form does insert fewer ϕ-functions compared to the naive alternative, which is to instantiate a ϕ-function for each variable at the entry point of each basic block in the program that has more than one predecessor.

- minimal better than "naive"
- transition for semi-pruned
(minimal not "minimal")

12 Semi-pruned SSA Form

instead give a definition and
an exemple

Semi-pruned SSA From, introduced by Briggs et al. [1], was based on the observation that many variables are never used outside of the basic blocks where they are defined. Consequently, there is no need to instantiate any ϕ-functions for any of them. Specifically, if every use of variable v occurs after a definition of v in the same basic block, then no ϕ-functions are necessary, although each definition and use must still be renamed. All of these variables are filtered out, and the algorithm to construct Minimal SSA Form is then applied to the remaining variables. For many applications, the vast majority of variables are filtered, which significantly reduces the number of ϕ-functions that are instantiated, and avoids the corresponding increase in the size of the symbol table to accommmodate variables that would otherwise be defined by ϕ-functions.

semi-pruned SSA construction

13 Liveness and Liveness Analysis

Say that it is useless to have phi functions at points where the variable is not live. Then say that liveness can be defined as follow. With this definition pruning non live phi is equivalent to removing dead phi (that can be performed easily as mentioned earlier).

Now, we turn our attention to variables and their lifetimes. By exploiting this information, we can further reduce the number of ϕ-functions instantiated during the conversion to SSA Form.

liveness for pruned

Variable v is *live* at some point p in a program if:

"definition" of is live

1. There is a path from some definition of v to p, and
2. There is a use of v reachable from p, meaning that there is a path from from p to the use of v that does not include a definition of v.

The *live range* of a variable is the set of points at which it is live. A variable must reside in a storage location, in a register, in memory or both, throughout its live range.

liverange

Liveness Analysis is the process of computing the live range of each variable in a procedure. The bulk of the work is performed by an iterative data flow analysis that computes either the *Live-In* or the *Live-Out* sets of each basic blocks, i.e., the set of variables that are live at the entry or exit of each basic block in the program. Given this information, a linear traversal of each basic block yields the exact set of variables that are live at each program point. In general, the sets of program points that constitute each variable's live range are not computed.

liveness analysis

out of the scope of this chapter

Here, we will show how to compute the Live-Out set of each basic block. $LIVEOUT[n]$ will denote the Live-Out set of basic block n. Each of the sets is represented as a bit-vector, with one bit corresponding to each variable. If V is the set of variables in the procedure, then each bit-vector contains $|V|$ bits. To simplify notation, we will associate each variable v_i with its index i, so $LIVEOUT[n][i] = 1$ if v_i is live at the exit point of basic block n, and $LIVEOUT[n][i] = 0$ otherwise.

To perform liveness analysis, we need two other sets for each basic block. $UEVAR[n]$ is defined to be the set of *upwards exposed* variables in basic block n, i.e., those variables that are used in n, but do not have a definition preceding them in n. Clearly, the variables belonging to $UEVAR[n]$ are live at the entry point of n; Liveness analysis is a backward dataflow analysis, as variable lifetimes are propagated backwards from $UEVAR[n]$ and into the predecessors of n.

$VARKILL[n]$ contains the set of variables that are defined in basic block n. As liveness analysis is a backward data flow analysis, the backward propagation of liveness information ends with the definition of a variable. The $VARKILL$ sets effectively stop the backward propagation of variables that are live at their definition points.

Initially, the sets $LIVEOUT[n]$ are empty for each basic block, and a linear traversal of each basic block yields the sets $UEVAR[n]$ and $VARKILL[n]$. Once the initial sets are established, the following data flow equations are repeatedly computed for each basic block n; this is called an *iteration*. The process repeats until an entire iteration fails to change any of the $LIVEOUT$ sets. At this point, stability is reached, and the information contained in the $LIVEOUT$ sets is wholly accurate.

[Insert liveness equations here]

The results of liveness analysis are generally used in three contexts: the construction of Pruned SSA Form, as described in the following section; register allocation; and the translation out of SSA Form.

use of liveness

14 Pruned SSA Form

Summaries. Give an example instead to illustrate the difference between pruned/semi-pruned/ minimal.

Minimal SSA Form does not take variable liveness information into account when instantiating ϕ-functions. Semi-pruned SSA recognizes that many variables are never live across the boundaries of the basic blocks that contain their definition, and does not instantiate any ϕ-functions for these variables. Under both Minimal and Semi-pruned SSA Form, the ϕ-functions that are instantiated are done so on the basis of iterated dominance frontiers alone, and do not account for any additional liveness information about the corresponding variable. Therefore, ϕ-functions are often instantiated for variables at the entry point of basic blocks where the variable is not actually live. *Pruned SSA Form*, in contrast, requires that ϕ-functions are only instantiated for a variable v at the entry point of basic blocks where v is live [5].

pruned SSA live @ entry point

There are two possible methods to construct Pruned SSA Form. The direct method is to perform liveness analysis upfront, and check whether a variable is live at the entry point of a basic block before instantiating a ϕ-function for that variable. This approach is straightforward, but requires performing liveness analysis upfront.

construction with liveness analysis

Briggs et al. [1], who introduced Semi-pruned SSA Form, compared the runtimes of constructing Minimal SSA, Semi-pruned SSA, and the direct method of constructing Pruned SSA Form. Minimal SSA Form instantiated an exorbitant number of ϕ-functions, and, as a consequence, it had the slowest runtime. Semi-pruned SSA Form was the fastest to construct, because Pruned SSA Form required liveness analysis in advance. Although Semi-pruned SSA Form instantiated significantly more ϕ-functions than Pruned SSA Form, the overhead of liveness analysis was significantly greater than the overhead of instantiating the extra ϕ-functions.

semi-pruned versus pruned. Construction time # of phis

It is also possible to construction Pruned SSA by first building Semi-Pruned SSA and then using a simplified dead code elimination algorithm on ϕ-functions to convert the resulting program into Pruned SSA Form; in fact, a more algorithm that eliminates dead code as well as dead ϕ functions also suffices. ~~To the best of our knowledge, such an algorithm has not been formally published; however, Choi et al. [5] suggested that such an algorithm may exist, and the method has even been patented [I'm not sure how to cite this].~~ ~~As long as the runtime of the dead code elimination step plus the cost of instantiating the extra ϕ-functions, which are later removed, is cheaper than performing liveness analysis, this latter method will run faster.~~

construction with dead-code

15 Live Range Insersection and Interference

Liveness information also plays an important role in register allocation. Any variable must reside in a storage location: a register, in memory, or both, at each point when it is live; otherwise, its value will be lost. The register allocator, therefore, must partition the variables between register and memory at each point in the program, and make sure that at most one variable is allocated to each register at any point.

register allocation

not always true: eg the double diamond: two variables share the samle register.

The live ranges of two variables *intersect* if there is at least one point in a procedure where both variables are live. In general, two variables whose live ranges intersect cannot share the same storage location; however, there is one exception: Chaitin et al. [4, 3] suggested that two variables whose live ranges overlap, but are known to always hold

interference =~ intersection

the live-range of one intersects the def of the other which is equivalent to intersection of live-ranges when the code is strict.
But considering the SSA tobe strict simplifies a lot the analysis

we can define interference
== intersect AND != value
So intersection still useful
We can approximate
the notion of value using
simple value numbering.

the same value, can share the same register; unfortunately, testing whether or not two computations produce the same result is undecideable, as it is a generalization of the *Halting Problem*. Therefore, Chaitin's more general notion of live range *interference* is often simplified to live range intersection, or intersection with provisions to account for constant values.

notion of value

Many register allocators build an auxiliary data structure called an *interference graph* to assist with the allocation process. An interference graph is denoted by the tuple $G = (V, E, A)$, where V is a set of vertices, one for each variable in the procedure, E is a set of *interference edges*, and A is a set of *affinity edges*. An interference edge (v_i, v_j) is placed between every pair of variables v_i and v_j that interfere with one another; in principle, any definition of interference that at least includes intersection can be used. An affinity edge (v_k, v_l) is placed between two variables that are repectively defined and used by a copy operation, i.e., $v_k \leftarrow v_l$ or $v_l \leftarrow v_k$; in this case, assigning v_k and v_l to the same register eliminates the copy operation. More details on this process will be provided in section ...

interference & affinity
graphs

dominance property allows
also the tree-scan (the
live-range contains no
gap on the dominator tree).
It allows also simple
tests such as dominance
test for reachablty/availblty
(usefull for code-motion
and PRE) it allows
simple forward propagation
using topological
order

An SSA Form program that is also strict satisfies a very important property: the definition point of each variable dominates each use [2]; from there, it is fairly straightforward to show that the definition point not only dominates each use, but the entire live range of the resulting variable. In other words, each live range is a subtree of the dominator tree. The resulting interference graph, therefore, is the intersection graph of a set of subtrees of a tree, which is precisely the definition of the class of *chordal* graphs. Chordal graphs are significant because several problems that are NP-complete for general graphs have efficient polynomial-time solutions for chordal graphs, including graph coloring, which is often used as a model for register allocation. This theory will be presented in much greater detail in future chapters of this book.

with dominance property
interference graph
(in fact intersection)
is chordal

16 Philip's comments

Detailed example (hopefully, corresponding to the procedure illustrated in the preceding section.

Here, I anticipate 1 page of text and 1 page for the illustration.

??

17 Conventional and Transformed SSA Form

The conversion to SSA form replaces each variable v in the pre-SSA program with a set of variables $v_1, v_2, \ldots, v_k$, which perfectly partition v. At every point in the procedure where v is live, *exactly* one variable v_i is also live; and none of the v_i are live at any point where v is not.

SSA = single reaching def

Based on this observation, we can partition the variables in a program that has been converted to SSA Form into congruence classes. We say that x and y are *ϕ-related* to one another if they are referenced by the same ϕ-function, i.e.: either both x and y are parameters of the ϕ-function, or, without loss of generality, x is a parameter, and y is defined by the ϕ-function. This relation is

SSA webs

1. *reflexive*: x is ϕ-related to x;

2. *symmetric*: x is ϕ-related to y if and only if y is ϕ-related to x; and
3. *transitive*: if x is ϕ-related to y and y is ϕ-related to z, then x is ϕ-related to z.

Therefore, this notion of ϕ-relationship is itself and equivalence relation, meaning that the transitive closure of the relation partitions the variables defined locally in the procedure into equivalence classes. By the reflexive property, each variable that is not involved in a ϕ-function belongs to a singleton class. Let ϕ-*CongruenceClassx* denote the equivalence class containing variable x.

The conversion to SSA Form, as described above, ensures that all varibles v_i introduced for pre-SSA variable v belong to the same ϕ- congruence class. The program can be immediately translated out of SSA Form by replacing each definition or use of v_i with v instead, and deleting the ϕ-functions. This simplistic reverse transformation is not always possible after SSA-based program transformations and optimizations such as dead code elimination, copy folding and propagation, and various forms of code motion are applied [1].

??

In particular, transformations applied to an SSA Form program may merge the ϕ-congruence classes of distinct variables u_i and v_j, which respectively correspond to pre-SSA variables u and v respectively. Moreover, u_i and v_j may interfere, making it impossible to replace all of the variables in a single ϕ-congruence class with a single variable.

not clear
this is the case of copy-folding

freshly constructed SSA
is conventional
Conventional = can rename
vi in to v safely
copy-prop, code motion
=> non-conventional

Note: A figure showing an example here would be very nice.

For example, suppose that the lifetimes of u_i and v_j overlap, and we replace all of the variables in their ϕ-congruence class with a new variable x during the translation out of SSA Form. Then when the program executes, there will be a point where x is live, and it will contain one value: either the value corresponding to u_i or v_j, but not both. This is incorrect, as both values are required, because they will be used at some point later in the program as per the definition of liveness. As a consequence, the semantics of the program have not been preserved during the translation out of SSA Form.

why conventional
cannot be renamed

Sreedhar et al. [7] defines a program to be in *Conventional SSA (CSSA)* if it is possible to replace all occurrences of variables in a ϕ-congruence class with a single variable without affecting program correctness; in other words, none of the variables in a ϕ-congruence class interfere with one another under CSSA Form. The basic SSA construction algorithms all build CSSA Form; however, CSSA Form is not preserved after numerous transformations and optimizations are applied. A program is defined to be *Transformed SSA (TSSA)* Form if at least two variables in a ϕ-congruence class interfere. The conversion from TSSA to CSSA Form is an important and required step in order to translate out of SSA Form. Efficient and effective algorithms to translate out of SSA Form will be presented in Sections ...

summary:
CSSA, TSSA definitions
from one to another

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

1. Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, 1998.

2. Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32, 2002.
3. G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982. ACM.
4. G.J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1980.
5. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, New York, NY, USA, 1991. ACM.
6. 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.
7. Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210, 1999.
8. Michael Weiss. The transitive closure of control dependence: the iterated join. *ACM Lett. Program. Lang. Syst.*, 1(2):178–190, 1992.
9. Michael Wolfe. J+=j. *SIGPLAN Not.*, 29(7):51–53, 1994.