Andrew Daniels

Writing Competency Requirement

Biola University

3 April 2021

Static Single Assignment Form: Brief History and Analysis

Abstract

This document explores several original thesis documents by the progenitors(?) of *static single assignment form*, some devising the algorithm, some improving it, and others demonstrating its effectiveness in code optimization problems. Static single assignment (SSA) form is a type of intermediate representation that's used to streamline a variety of code optimization algorithms, including but not limited to constant propagation and dead code elimination. This paper looks at Kildall's simple constant propagation as an introduction to a common code optimization problem, then introduces the first thesis ever to demonstrate SSA form, and then some of the algorithmic improvements made by other researchers years after. The last section is a brief analysis of Wegman and Zadeck's sparse conditional constant propagation algorithm, which uses SSA form to improve on Kildall's original algorithm. Read this as a report that gleans a basic understanding of what is contained in these theses. It goes into depth on a number of data structures: lattices, control flow graphs, value graphs, dominance frontiers, dominator trees, and def-use chains.

Introduction

Static single assignment form is a type of intermediate representation that is used to make code optimization simpler to compute. Called SSA form, it focuses on eliminating the complexity that is introduced to a program when multiple loads and stores are called on the same memory binding. In a way, IR code using SSA form reinterprets a program as a pure functional program, replacing each assignment to each variable in the original code with a unique immutable constant to which a value can be assigned only once.

A precursor to SSA form, and one that helps gain a beginner's understanding of how it and some of the following algorithms work, is Kildall's lattice framework for dataflow analysis. It helps introduce the problem of constant propagation, which SSA form can be used to improve drastically, as we'll see in Wegman et al. Alpern's thesis will introduce the basic of idea of SSA form itself, followed by some algorithmic improvements, once by Cytron et al. and then by Aycock and Horspool; and last, Wegman's algorithm.

Kildall's Lattice-theoretic Framework

In 1976, Kildall submitted a generalized algorithm for use in dataflow analysis problems. By this time, algorithms had been developed for a number of different code optimization problems, including constant propagation, which eliminates common subexpressions and redundant load operations, and live variable analysis (called "live expression" analysis in Kildall's text) (Kildall 194).

Horwitz provides a detailed explanation of Kildall's algorithm that focuses on an example of its use in connection with constant propagation and live variable analysis. This paper will focus on the constant propagation algorithm. This section follows Kildall's work, but Horwitz was used for clarification.

Theory

In his paper, Kildall describes a program's code in the form of a single data set and a sequence of statements that each change the elements in the data set. Kildall effectively identifies what most dataflow problems have in common and breaks it up into four axioms:

Axiom 1. The variables in a program together form a program state which is subject to change over the program's course, written as an ordered tuple,

$$D \coloneqq (d_1 , d_2 , d_3 , \dots , d_m) \tag{1}$$

where m is the number of variables in the program.

Axiom 2. The facts that need to be recorded for analysis, for each variable in the program, belong to a domain of values that form a partially ordered set.

$$[\ \forall d \in D\ ](\ d \in S\ ) \tag{2}$$

$$[\ \forall D\ ](\ D \in S^m\ ) \tag{3}$$

$$S \coloneqq \{\ \top, s_1 , s_2 , s_3 , \dots , s_n , \bot\ \} \tag{4}$$

$$\top \prec s_1 \preceq s_2 \preceq s_3 \preceq \dots \preceq s_n \prec \bot \tag{5}$$

$\top \coloneqq$ an arbitrary *top* element, sometimes written as 0

$\bot \coloneqq$ an arbitrary *bottom* element, sometimes written as 1

Axiom 3. Each statement, or collection of statements, in the program is expected to change the program state in some way, and thus can be represented in the form of a function that maps from the domain of all program states to the domain of all program states.

$$[\ \forall k\ ](f_k : S^m \rightarrow S^m\ ) \tag{6}$$

$$\langle\ \text{Line } k : \text{assign } v_i \coloneqq c_j\ \rangle \quad \Rightarrow \quad f_k(D) \coloneqq (D - \{\ (v_i , *)\ \}) \cup \{\ (v_i , c_j)\ \} \tag{7}$$

Axiom 4. The convergence of two dataflow paths (e.g. the end of a branching statement), and thus two sets of facts, is resolved using a “meet” operation, which represents the least upper bound of two given values.

Points (2) and (4) above describe the properties of a lattice: a partially ordered set and a meet operation. In a partially ordered set (poset), any two elements have a partial ordering property (≤), and in a lattice, every two values in the poset have a least upper bound that is also in the set. Given lattice L, poset S, and meet operation $\sqcap$:

$$L \coloneqq (S, \sqcap) \tag{8}$$

$$[\ \forall x, y \in S\ ](\ \exists w \in S \ni w = x \sqcap y\ \wedge\ x \leq w\ \wedge\ y \leq w\ ) \tag{9}$$

For any x and y in S, there exists w in S such that w is the meet of x and y, and w is at least x and at least y.

$$[\ \forall x, y, w, z \in S\ ](\ x \sqcap y = w\ \wedge\ x \sqcap y = z\ \Leftrightarrow\ w = z\ ) \tag{10}$$

The meet of any two elements is unique: only one element can be the least upper bound of two elements.

$$[\ \forall x \in S\ ](\ x \sqcap \top = \top \sqcap x = x\ ) \tag{11}$$

$$[\ \forall x \in S\ ](\ x \sqcap \bot = \bot \sqcap\ \ x = \bot\ ) \tag{12}$$

Arithmetic of lattice extrema; $\top$ is annihilated by all lattice elements in a meet; $\bot$ annihilates all lattice elements in a meet.

If a lattice is "complete," it also has a "join" operation, such that every two elements has a greatest lower bound as well as a least upper bound. The "join" operation ($\sqcup$) is a mathematical dual of the "meet" ($\sqcap$).

Algorithm

Kildall starts by converting the program itself into a directed graph, in the vein of most previously explored dataflow algorithms at the time. This is called a control flow graph (CFG), given here as $CFG(P)$, (i.e. the control flow graph of the program). Each individual code statement—namely, each statement that either branches the control flow or makes a change to a variable's value—is represented by a node in the graph, and the flow of control from one statement to the next is an edge in the graph.

```
1:   U ← 10
2:   V ← 20
3:   while P
4:       if Q
5:           U ← U + 2
6:           V ← V + 3
7:       else
8:           U ← U + 1
9:           V ← V + 2
10:  U ← 10 · U
11:  V ← 5 · V
```

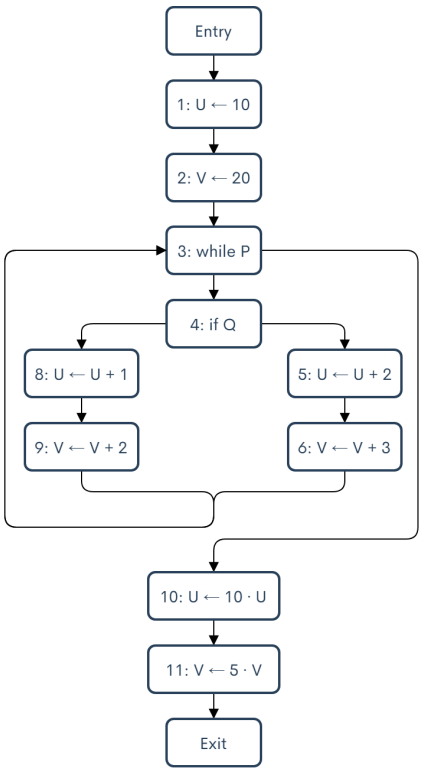


Figure 1. The flowchart on the right represents the control flow graph of the program to the left. [Author's note: All figures in this paper are original examples, made using zenflowchart.com.]

According to theory (provided by Horwitz), in order for Kildall's algorithm to achieve a conservative solution (one that closely approximates the desired result), the dataflow functions used must be at least monotonic:

$$[\ \forall x, y \in S\ ](\ x \preceq y\ \Leftrightarrow\ f_k(x) \preceq f_k(y)\ ) \tag{13}$$

This means that, for an optimistic problem like constant propagation, which assumes every binding to be constant until proven otherwise, a dataflow function cannot produce *more* constant variables than it receives, and for a pessimistic problem like live variable analysis, which assumes every binding to be dead until proven otherwise, a dataflow function cannot produce *fewer* live variables than it receives.

The ultimate goal, then, of a code-optimizing algorithm, using Kildall's framework, is to find a meet-over-all-paths solution that represents the maximal fixed point solution. "Meet-over-all-paths" means that each node's corresponding function has been applied to the program's data set in control order exactly once, and the designated combining operation (typically the meet operation mentioned earlier) has been applied to all converging sets of data exactly once, so that each program statement has been visited by the algorithm exactly once (McKinley). A solution is "maximal" means that the solution contains all data points (e.g. constants or live variables, as this is what the optimizer is looking for) that can possibly be found in the program code, and the data set is at a "fixed point" means that the application of some function no longer has an effect on the data (e.g. iterating the algorithm will produce no more constants or live variables), and thus we've found all that this algorithm can possibly find (McKinley).

In order for the meet-over-all-paths solution (a run-through of the algorithm) to be guaranteed equivalent to the maximal fixed point solution (the ultimate goal of the algorithm), all dataflow functions must distribute over the meet operation (McKinley). If they do not necessarily distribute, then any application of a dataflow function could be seen as corrupting the output of all subsequent meet operations and thus decreasing the accuracy of the final result.

$$[\ \forall x, y \in S\ ][\ \forall n_k \in CFG(P)\ ](f_k(x \sqcap y) = f_k(x) \sqcap f_k(y)\ ) \qquad (14)$$

For each node in the control flow graph of the program, the corresponding dataflow function distributes over the meet operation.

Example: Constant Propagation

Constant propagation identifies all variables in a program that evaluate to constants at compile time. For this type of algorithm, each statement in the program is assigned a *before*-data set and an *after*-data set, meaning, what the data set looks like *before* and *after* the statement's execution.

1: for each statement $n_k \in CFG(P)$

2: $before(D_k) \leftarrow \varnothing$

3: $after(D_k) \leftarrow \varnothing$

4: for each statement $n_k \in CFG(P)$

5: $before(D_k) \leftarrow \prod_{u \in Pred(D_k)} after(u)$

6: for each $(v, c) \in before(D_k)$

7: replace v with c in $RHS(n_k)$

8: $after(D_k) \leftarrow before(D_k)$

9: for $\langle v \leftarrow LHS \rangle \in$ statement n_k

10: $temp \leftarrow Evaluate(RHS)$

11: if $\top < temp < \bot$

12: $after(D_k) \leftarrow (after(D_k) - \{ (v, *) \}) \cup \{ (v, temp) \}$

Algorithm 1. Adapted from the notes by Horwitz on constant propagation.

Line 1: $CFG(P)$ represents the control flow graph of the program P, which contains a node n_k for each program statement, an additional entry node n_0, and the control flow edges between them.

Line 5: The meet operation, applied over all of the current node's predecessors. The $\prod$ represents an aggregate form of the *general* framework meet operation. Due to its optimistic approach, the particular operation for constant propagation is *set intersect.* Rather, if a variable has unmatching constant values across multiple sets, it's ignored, because it's not a constant. Conceptually, this is the same as assigning $\bot$.

Lines 6–12: These lines represent the dataflow function f_k: replace each *use* of a constant variable with its currently discovered constant value, then attempt to evaluate the right-hand side of the statement; if the evaluation successfully returns a constant (i.e. $\top < temp < \bot$, whereas $\top$ is undefined

and $\perp$ means *proven to vary* and thus not constant), then replace the current ordered pair $(v, *)$ with the new $(v, temp)$.

Pseudocode	Propagated Constants
1: $a \leftarrow 18$	$after(D_0) = \{ \}$
2: $b \leftarrow 9 - (a / 3)$	$after(D_1) = \{ (a, 18) \}$
3: $c \leftarrow b \cdot 4$	$after(D_2) = \{ (a, 18), (b, 3) \}$
4: if Q	$after(D_3) = \{ (a, 18), (b, 3), (c, 12) \}$
5: $c \leftarrow c - 10$	branch: $after(D_5) = \{ (a, 18), (b, 3), (c, 2) \}$
6: else	
7: $c \leftarrow c + 10$	branch: $after(D_7) = \{ (a, 18), (b, 3), (c, 22) \}$
8: return $c \cdot (60 / a)$	meet:

$$before(D_8) = after(D_5) \sqcap after(D_7)$$

$$= \{ (a, 18), (b, 3), (c, 2) \} \cap \{ (a, 18), (b, 3), (c, 22) \}$$

$$= \{ (a, 18), (b, 3) \}$$

$$after(D_8) = \{ (a, 18), (b, 3) \}$$

Note: The above solution is a simplification of the ordered-tuple form of the solution, given in Axiom 1 of Kildall's framework specifications:

Algorithm data	Formal Kildall representation
	$a \quad b \quad c$
$\varnothing$	$(\top, \top, \top)$
$\{ (a, 18), (b, 3) \}$	$(18, 3, \bot)$

Kildall's algorithm is naïve, namely in that it doesn't take control logic into account. For instance, if the control predicate Q on Line 4 above were replaced with $c < 10$, Kildall's algorithm would fail to notice that Line 7 would never execute and thus eliminate the meet conflict. For now, however, $(18, 3, \bot)$ is considered the maximal solution, since the naïve form of this algorithm doesn't try to predict branch decisions. Later we'll see a version of constant propagation that can detect dead branches.

Analysis

Upon inspection, lines 4 and 6 are the source of the algorithm's worst-case complexity, since it's possible that Line 7 could run a total of $k \cdot |V|$ times, k being the number of statements and V the set of variables in the program, and Line 7 could run a total of $|V|$ code replacements each time it executes. This subsumes all other subprocesses in the algorithm in terms of complexity, making Kildall's Simple Constant Propagation algorithm $O(|N| \cdot |V|^2)$, N being the set of nodes (or statements) in the program code. This confers with Wegman and Zadeck's later analysis of the algorithm (Wegman 187).

Alpern et al.'s Algorithm for Programatic Equivalence

In 1988, Alpern, Wegman, and Zadeck published a paper on "programmatic equivalence". Their original intent was to introduce a method for identifying redundant code using a form of value numbering. In value numbering, unique names are given to individual assignments to variables, rather than to variables themselves. Wegman, Zadeck, and Barry Rosen would go on to formalize this algorithm in another paper the same year ("Global value numbers and redundant computations"), calling it the static single assignment form as it relates to intermediate representation. This paper will focus on the former thesis, which introduced the guiding concept.

Value Numbering

In the value-numbering step of Alpern et al.'s algorithm, an index is applied to each instance of each variable according to how many times it has had a new value assigned to it, and all subsequent uses of the variable are replaced with the latest indexed name.

Example

	Pseudocode	Value-numbered Pseudocode
1:	$B \leftarrow 5$	$B_1 \leftarrow 5$
2:	$A \leftarrow 10$	$A_1 \leftarrow 10$
3:	$B \leftarrow A + 3$	$B_2 \leftarrow A_1 + 3$
4:	$C \leftarrow B \cdot 5$	$C_1 \leftarrow B_2 \cdot 5$
5:	$D \leftarrow (A \cdot 5) + 15$	$D_1 \leftarrow (A_1 \cdot 5) + 15$
6:		

According to Alpern et al., value numbering methods in use at the time of writing would have been based on symbolic equivalence, which would fail to detect instances where different code statements have equivalent evaluations (1). For instance, symbolic equivalence identifies that Line 1 is dead code, because value B_1 is never used past that point in the program, but it fails to identify that C_1 and D_1 are equivalent by Line 6. Alpern et al.'s algorithm demonstrates how to determine this type of algorithmic equivalence using value numbering in a general dataflow problem.

Alpern et al.'s algorithm follows Kildall's except in the fact the program CFG is reduced to the point that the only edges in the graph are the ends of branching code. The nodes in this type of graph are referred to as basic blocks (2). A basic block in a CFG contains all contiguous statements not including branching edges.

```
1:    U ← 10
2:    V ← 20
3:    while P
4:        if Q
5:            U ← U + 2
6:            V ← V + 3
7:        else
8:            U ← U + 1
9:            V ← V + 2
10:   U ← 10 · U
11:   V ← 5 · V
```

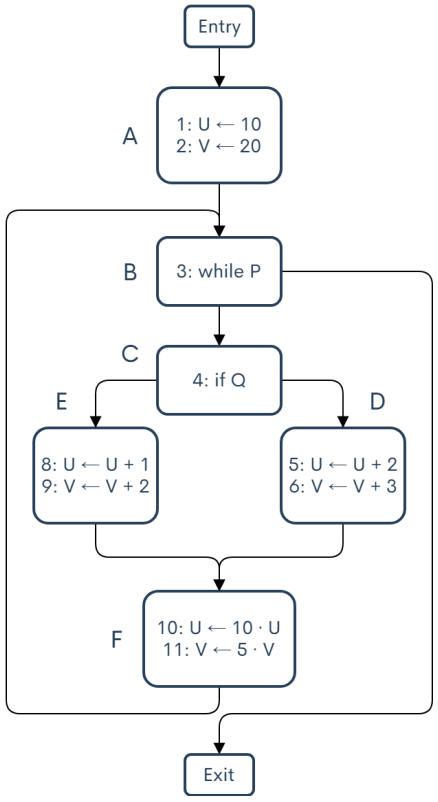


Figure 2. The same program from Figure 1 and its control flow graph to the right. Ignoring the *Entry* and *Exit* nodes, the graph is irreducible: it contains only basic blocks, which means all edges are branching edges. Note that edges converge at the end of the if-statement at Node F and at the top of the while-statement at Node B. This is where join points are located.

When a node in the CFG has multiple predecessors, it's labeled a join node (2). This shouldn't be confused with the "join" in Kildall's framework. In Kildall, *meet* operations occur at nodes that have multiple predecessors and *join* operations typically occur at nodes that have multiple successors, but in theory, which operation occurs where, depends not on the properties of the nodes themselves but on the lattice values in the data set (Horwitz).

Phi Functions

Notice in the graph in Figure 2, nodes D and E introduce conflict for variables U and V. The node that follows them, Node F, uses both U and V, but which versions of each cannot be determined. If Q

evaluates to True, then the assignments in Node D will be used in Node F, but if Q evaluates to False, then the assignments in Node E will be used instead. Alpern et al. resolve this with the use of a ϕ-function:

$$a_3 \leftarrow \phi(a_1 , a_2)$$

The above statement means that, if control reaches this statement from the first branch (whichever branch is the first branch), then assign the value in a_1 to a_3, else assign a_2 to a_3. As coming examples will show, the node conflicts which ϕ-function are being used to resolve seem to occur exclusively as the result of a conditional construct, such as if and while, so Alpern et al. use a variant of the ϕ-function that includes the control predicate as its first argument:

$$a_3 \leftarrow \phi(Q, a_1 , a_2)$$

and has the meaning that, if Q evaluates to True, then assign the value in a_1 to a_3, else assign a_2 to a_3, implying that control would have to reach this statement from the "True"-branch in order for a_3 to be assigned a_1's value, etc (3).

1: $U_1 \leftarrow 10$

2: $V_1 \leftarrow 20$

2a: $U_2 \leftarrow U_1$

2b: $V_2 \leftarrow V_1$

3: while P

4: if Q

5: $U_3 \leftarrow U_2 + 2$

6: $V_3 \leftarrow V_2 + 3$

7: else

8: $U_4 \leftarrow U_2 + 1$

9: $V_4 \leftarrow V_2 + 2$

9a: $U_5 \leftarrow \phi(Q, U_3, U_4)$

9b: $V_5 \leftarrow \phi(Q, V_3, V_4)$

10: $U_6 \leftarrow 10 \cdot U_5$

11: $V_6 \leftarrow 5 \cdot V_5$

11a: $U_2 \leftarrow \phi(P, U_1, U_6)$

11b: $V_2 \leftarrow \phi(P, V_1, V_6)$

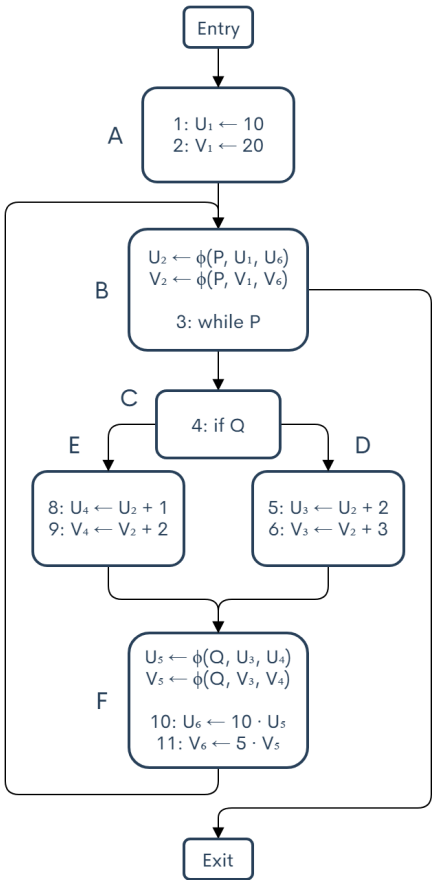


Figure 3. The same program from Figure 1 in static single assignment form and its control flow graph to the right. Value-numbering conflicts are temporarily resolved using ϕ-functions in the added lines (2a, 2b, 9a, 9b, 11a, and 11b).

Value Graphs

The goal of Alpern et al.'s thesis was to show how static single assignment form reveals algorithmic equivalence, which is a necessary step to detect redundant code statements (4). To fully accomplish this, they introduce value graphs: for each static-single-assigned value in the IR code,

construct a graph that will propagate over a traversal of the CFG. Whenever a value is assigned to a variable, add the value to the variable's value graph in the form of a syntax tree. (Since the CFG is in SSA form, this is expected to happen only once per variable.) Each node in the graph represents a function with a directed edge to each of its arguments. This includes the assignment operation. For example, the complete value graph of the statements $\langle\, x \leftarrow y + 9 \,\rangle$ and $\langle\, y \leftarrow 2y \,\rangle$ would be written as

$\{\, (x \rightarrow \text{add}), (\text{add} \rightarrow y), (\text{add} \rightarrow 9), (y \rightarrow \text{mult}), (\text{mult} \rightarrow 2), (\text{mult} \rightarrow y) \,\}$.

If two different value graphs at the *same point* in the program code are *congruent*, then the corresponding values are determined to be equivalent (4). This process can be used to resolve ϕ-functions completely, which up to this point in the algorithm have acted as placeholders for value-numbering conflicts.

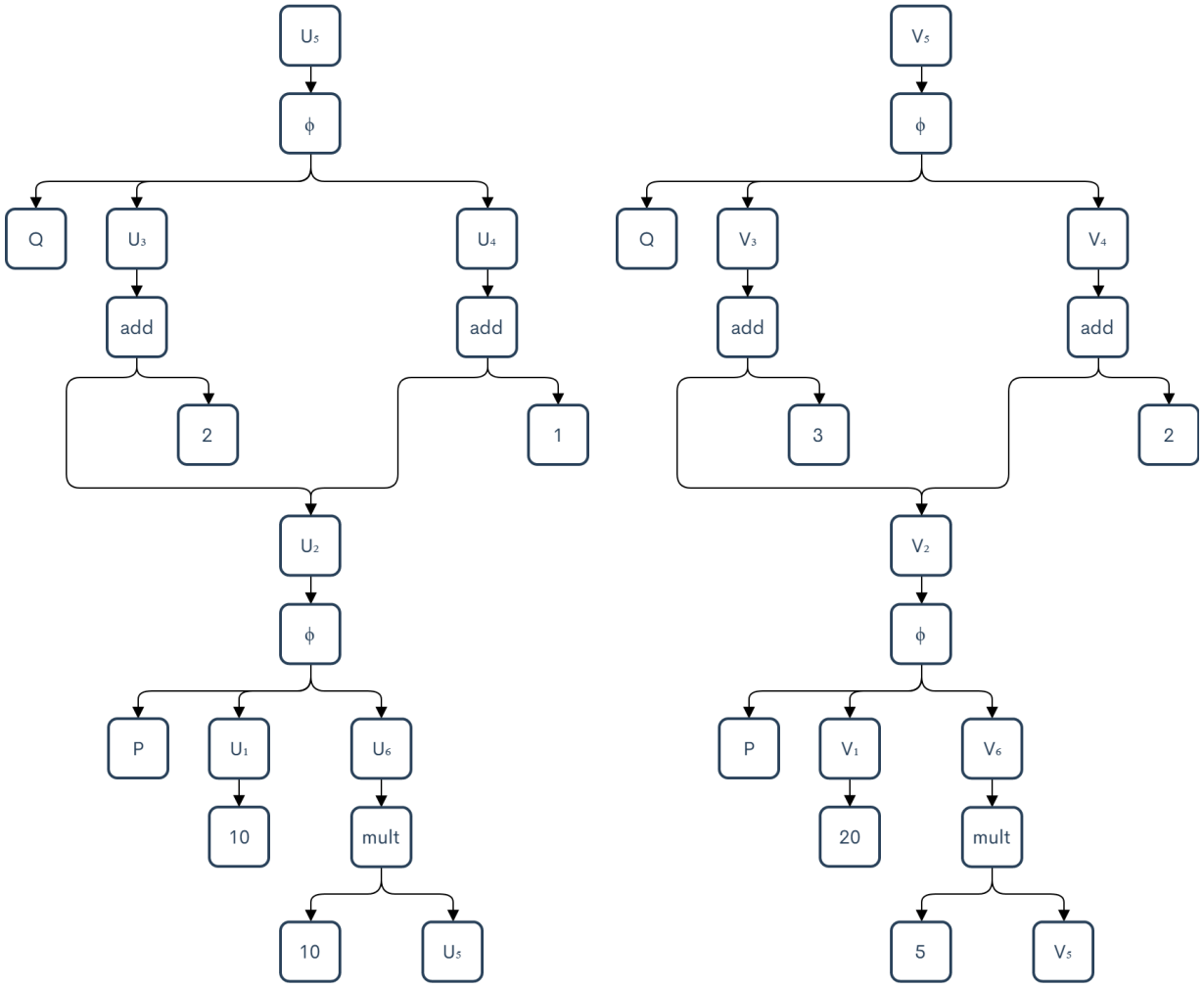


Figure 4. Value graph representations of U_5 and V_5 from Figure 3 at Line 10 in the program code. Each node is treated as a function and points to its arguments. (Each value name is treated as a function of its assigned value.) In this example, U_5 and V_5 have very similar value graphs. Notice the subgraphs for U_3 and V_3. If Line 6 were changed to $\langle\ V_3 \leftarrow V_2 + 2\ \rangle$ in the program code in Figure 3, then their value graphs would be congruent, and since they occur in the same basic block, U_3 could replace all uses of V_3 in the IR code.

Redundant code elimination isn't the only code optimization problem improved by the use of SSA form. Later research would discover its general benefits in the field of code optimization, particularly, as we'll see later, in constant propagation (Cytron 452).

Cytron, Ferrante, Rosen, and Wegman: Minimal SSA Form

In 1991, Cytron et al. introduced their algorithms for more-robust and -efficient uses of SSA form, using the concept of control flow dominance (Cytron 457). In former SSA-building algorithms, a first step would be to add "trivial ϕ-functions" at each node for each variable in the program, and then going back later to remove the functions that have only one argument (e.g. $\langle\, a_3 \leftarrow \phi(a_3) \,\rangle$) (459). Cytron et al. introduce minimal SSA form, which uses a minimal number of ϕ-functions, and pruned SSA form, which omits ϕ-functions of variables from join points where no use of those variables follow (459). Cytron et al. recommend minimal over pruned as the more efficient algorithm, when exact conflict resolution isn't needed, and this paper only focuses on the former (459).

Dominance Frontiers

Building minimal SSA form relies on the concept of a dominance frontier, which in turn relies on the concept of control flow dominance (455). In control flow dominance, a node in a control flow graph is said to dominate another when control must always cross the former node to reach the latter. The dominance frontier of a node, then, is the set of nodes only just beyond the nodes it dominates, rather, the immediate successors of its dominated nodes.

Given a CFG with nodes X and Y, if a node X appears on every path from Entry to Y, and only then, then X is said to dominate Y. Thus, in order to get to Y, control must flow through each of its dominators. For theoretical purposes, nodes are allowed to dominate themselves, and all other dominators are called strict dominators.

$$X \operatorname{dom} Y \Leftrightarrow [\ \forall p \in CFG_{\ni} \ p : n_0 \rightarrow \ldots \rightarrow Y\ ](\ X \in p\ ) \tag{15}$$

X dominates Y means that X appears on every path in the CFG from the root to Y.

Control cannot reach Y without passing through each of its dominators.

$$[\ \forall X\ ](\ X \operatorname{dom} X\ ) \tag{16}$$

Domination is reflexive.

$$X \text{ dom } Y \wedge Y \text{ dom } Z \Rightarrow X \text{ dom } Z \quad (17)$$

Domination is transitive.

$$X \text{ domm } Y \Leftrightarrow X \text{ dom } Y \wedge X \neq Y \quad (18)$$

X strictly dominates Y means that X dominates Y, and X is not Y.

$$DF(X) = \{ Y \mid (\exists P \in Pred(Y))(X \text{ dom } P \wedge \neg(X \text{ domm } Y)) \} \quad (19)$$

The dominance frontier of a node X is the set of all nodes, Y, such that X dominates some predecessor of Y but does not strictly dominate Y. In other words, consider all the nodes X dominates; Y is in the *frontier* of one of those nodes.

Cytron et al.'s Theorem

The crux of Cytron's article is in proving that finding the dominance frontier of a control flow node is equivalent to finding its join points (467). The proof is not straightforward but can be described intuitively.

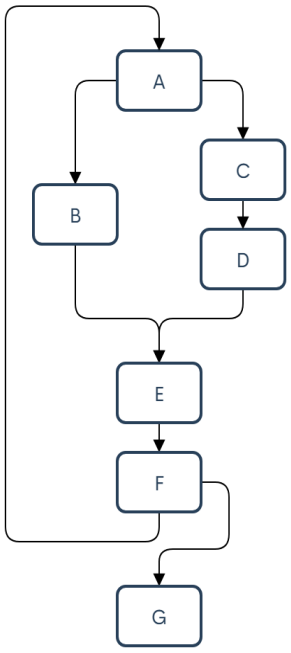


Figure 5. Intuition for Cytron et al.'s theorm.

In the figure above, Node C dominates D, because control must flow through C in order to reach D, and, in principle, node D dominates itself. Node E immediately follows these nodes, putting it in the frontier of the nodes dominated by C and D (and B for the same reasons), but control does not necessarily pass through C or D to reach E. Thus, E is in the dominance frontier of nodes B, C, and D.

Likewise, Node E dominates F, and F dominates F, but neither dominates A, because control could pass from F back up to A or continue down from F to G. Thus, A is in the dominance frontier of both E and F.

Cytron et al.'s theorem states that this relationship only ever occurs at join points. In a control flow graph, a node's dominance frontier is equivalent to its join point with another node, and finding the dominance frontier is a fast way of detecting this relationship.

Cytron et al.'s Algorithm

Cytron et al.'s algorithm uses a dominator tree, which identifies the dominance relations of all the nodes in the CFG. In their own literature, the words *predecessor*, *successor*, and *path* always refer to the CFG. The words *parent*, *child*, *ancestor*, and *descendant* always refer to the dominator tree (464). This can be confusing to a reader who's not paying attention. The parts of the algorithm that mention a node's "children" imply that the dominance relations in the CFG have already been established and recorded in a separate data structure.

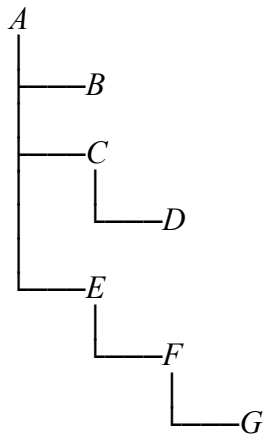


Figure 6. The dominator tree of the *CFG* example in Figure 5. Each node indicates which nodes it dominates in the *CFG* by branching down to them in the tree.

By the time of Cytron's writing, algorithms for building dominator trees had been developed which run in near-linear and linear times, but neither algorithm is straightforward (464). Cytron et al. only mention them in passing. This is to show that detecting dominance relations was a simple matter at the time and wouldn't contribute significantly to the algorithm's complexity (at least, for large programs).

The subroutine used to build the dominance frontier, for some node in a *CFG*, starts by considering all nodes that immediately follow the node itself, then recurses over its children in the readily constructed dominator tree (465).

$$DF(X) = DF_{local}(X) \cup \bigcup_{Z \in Children(X)} DF_{up}(Z) \tag{20}$$

$$DF_{local}(X) \stackrel{\text{def}}{=} \{\, Y \in Succ(X) \mid \neg(X \text{ domm } Y) \,\} \tag{21}$$

Due to the reflexive property of domination, any successor that X does not dominate is in the dominance frontier.

$$DF_{up}(Z) \stackrel{\text{def}}{=} \{\, Y \in DF(Z) \mid \neg(idom(Z) \text{ domm } Y) \,\} \tag{22}$$

Due to the transitive property of domination, the dominance frontier of nodes that X does strictly dominate may also contribute to the dominance frontier. The function *idom*

refers to the immediate dominator. Strict dominance is the transitive closure of immediate dominance.

Pseudocode

```
1:  DF(X: node in CFG):
2:      S ← ∅
3:      for each node Y ∈ Succ(X)                    {{ DF_local }}
4:          if idom(Y) ≠ X
5:              S ← S ∪ { Y }
6:      for each node Z ∈ Children(X)                {{ DF_up }}
7:          for each Y ∈ DF(Z)
8:              if idom(Y) ≠ X
9:                  S ← S ∪ { Y }
10:     return S
```

```
11:   PlacePhiFunctions(CFG):
12:       for each node X ∈ CFG
13:           HasPhiAlready(X) ← 0
14:           LastWorklisted(X) ← 0
15:       W ← ∅
17:       for each variable V, by number k
19:           for each node X ∈ CFG that has assignments to V
20:               W ← W ∪ { X }
21:               LastWorklisted(X) ← k
22:           while W ≠ ∅
23:               take X from W
24:               for each Y ∈ DF(X)
25:                   if HasPhiAlready(Y) < k
26:                       place ⟨ V ← φ( ) ⟩ at Y          {{ The operative line. }}
27:                       HasPhiAlready(Y) ← k
28:                       if LastWorklisted(Y) < k
29:                           W ← W ∪ { Y }
30:                           LastWorklisted(Y) ← k
```

Algorithms 2 and 3. Adapted from Cytron et al., pages 466 and 470, respectively.

Analysis

With the use of dominance frontiers, the work of finding appropriate ϕ-function placements in SSA form is made slightly faster. Cytron et al. point out that Line 23 will execute as many times as there are assignment statements in the program (A_{tot}), including any statements added to the IR by Line 26 (Cytron 471). Then, for each time a node X is taken from the worklist, every node in $DF(X)$ is polled. Thus, the operative line is expected to execute a total number of times equivalent to

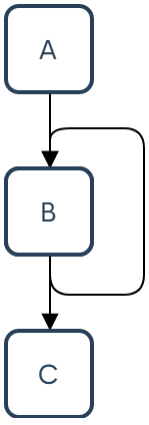
$$\sum_{X \in CFG} A_{tot}(X) \cdot |DF(X)|$$.

Upon inspection, dominance frontiers are very small for each node, so Cytron et al. report an average time complexity of $\Theta(A_{tot})$ for the combined work it takes to process the placement of ϕ-functions (471). A naïve algorithm by contrast, which runs in $O(N^2)$ time, would mark all nodes that can be reached from each assignment, instead of just the necessary nodes (Alpern 3).

Aycock and Horspool

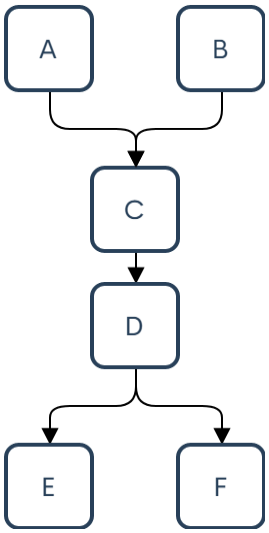
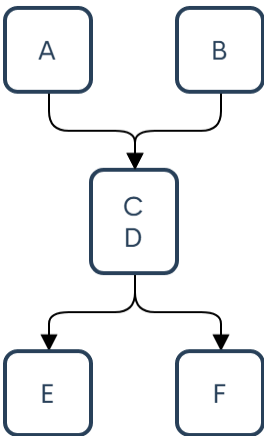
Aycock and Horspool would later supplant Cytron et al.'s algorithm in 2000 by proving that a control flow graph does not need to be fully reduced to basic blocks in order to achieve the same minimizing effect, since the only transformations needed to reduce the graph are the removal of self-edges and trivial nodes, but this is equivalent to the removal of two particular types of trivial ϕ-functions, a removal process which is less costly than graph transformation (Aycock 112).

Self-edge reduction

	Non-reduced form	Reduced form
Graphical meaning		

Algorithmic meaning	$\langle V_i \leftarrow \phi(V_i, \ldots, V_i) \rangle$	$\langle\rangle$

Trivial node reduction

	Non-reduced form	Reduced form
Graphical meaning		
Algorithmic meaning	$\langle V_i \leftarrow \phi(V_{x1}, V_{x2}, \ldots, V_{xk}) \rangle$ where $x_1, x_2, \ldots, x_k \in \{i, j\}$	$\langle V_i \leftarrow V_j \rangle$

If a ϕ-statement contains only arguments with the same identity as the left-hand side, then the statement is a self-edge and can be removed from the CFG completely. If a ϕ-statement contains only arguments that have one of two indices, one of them matching the index of the left-hand side the other not, then the statement is a trivial edge, and the entire ϕ-function expression can be replaced with an instance of the latter argument.

Wegman and Zadeck's Sparse Conditional Constant Propagation

Wegman and Zadeck's 1991 sparse conditional constant algorithm is the fourth iteration in a sequence of improvements on the constant propagation algorithm, listed in their paper and starting with Kildall, and is an example of how SSA form can be used to improve the scope and efficiency of a dataflow algorithm (Wegman 186). Wegman and Zadeck combine the work of two previous improvements on Kildall, one by Reif and Lewis that runs more efficiently, and the other based on Wegbreit's algorithm, which propagates more constants by detecting dead branches, something that, as stated before, Kildall's algorithm cannot do (186).

Def-use Chains

The Wegman and Zadeck's algorithm makes use of a data structure that's closely related to SSA form, called a def-use chain (189). A def-use chain, or def-use graph, can be identified as all ordered pairs of definitions and their uses, for all named values in a program. This structure is used to identify where a mentioned value is first defined and conversely all instances of where a defined value is mentioned.

$$DefUse(V) \coloneqq \{ (Def_V, Use_V) \}$$

$$Def_V \coloneqq \text{node containing a statement in which } V \text{ is defined}$$

$$Use_V \coloneqq \text{node containing a statement in which } V \text{ is used}$$

In SSA form, all Def_V are the same for each variable, since every value can have only one definition:

$$Def_V \neq Def_U \Rightarrow V \neq U \tag{23}$$

Rather:

$$DefUse(V)_{Def} \neq DefUse(U)_{Def} \Rightarrow V \neq U \tag{24}$$

In Wegman and Zadeck's algorithm, these ordered pairs are drawn directly onto the CFG itself as edges in the graph and marked as "SSA edges" (189).

Constructing def-use chains is trivial for a CFG using minimal SSA form and can be achieved in linear time (190). This cost is subsumed by the time taken to build the minimal SSA form of the CFG, which, as stated above, could take $O(E \quad V)$ or $O(E\ \alpha(E))$ (near-linear) time, depending on the approach.

Algorithm

The algorithm below is a particular interpretation of Wegman and Zadeck's Sparse Conditional Constant Algorithm using the notes in their thesis (Wegman 192). Their description of the algorithm is actually much more vague than the version given in this paper. The version provided here is given as a possible interpretation of those notes.

Definitions

In order to get a good understanding of the algorithm, think of each node in the CFG as an ordered quadruple: an array of lattice elements (one for each variable in SSA form), a single code statement, the list of inedges to the node, and the list of outedges to the node. Nodes in the CFG will be identified as the destinations of edges. Assume that for each graph constructed in this algorithm, each non-entry node contains exactly one code statement, so that the destination of each edge represents a single code statement.

$[\ \forall$ edge $E\ ](\ Dest(E) \coloneqq (\ \{\ LatticeCell(E, V_i)\ \}_{\forall i}, S, In, Out\ )\ )$

$\{\ LatticeCell(E, V_i)\ \}_{\forall i} \coloneqq$ an array of lattice elements; one for each static single assignment variable in the program; each edge has an array associated with it

$S \coloneqq$ a code statement, in the form of a conditional control structure

$\langle\ \langle$ if | while $\rangle\ Q\ \rangle$

or an assignment statement

$$\langle\, LHS \leftarrow RHS \,\rangle$$

$$In \coloneqq \text{list of inedges to } Dest(E), \text{ which includes } E$$

$$Out \coloneqq \text{list of outedges from } Dest(E)$$

Note that throughout the pseudocode, a code statement refers to either of the two cases that could describe S above, whereas a code expression refers only to a predicate (Q above) or the right-hand side of an assignment (RHS above). Some subroutines in the algorithm take code statements as input while others take code expressions as input. This was to make the pseudocode more rigorous, since Wegman and Zadeck don't distinguish the two.

Pseudocode

```
 1:  SparseConditionalConstant(P: program):
*2:      get CFG(P) in SSA form
*3:      for each variable V ∈ CFG(P)
 4:          CFG(P) ← CFG(P) ∪ DefUse(V)
*5:      flow ← Out( Entry(P) )
 6:      ssa ← ∅
 7:      for each edge E ∈ CFG(P)
 8:          Executable(E) ← False
 9:          for each variable V ∈ P
10:              LatticeCell(E, V) ← ⊤
11:      while flow ≠ ∅ ∧ ssa ≠ ∅
*12:          let E ∈ flow ∪ ssa
13:          if E ∈ flow
14:              flow ← flow − { E }
15:              if ¬Executable(E)
16:                  Executable(E) ← True
```

```
17:                     if Dest(E) match ⟨ V ← φ(W) ⟩
18:                         VisitPhi(E, Dest(E))
19:                     if Dest(E) has exactly 1 executable inedge
20:                         VisitStatement(E, Dest(E))
* 20a:                  if Dest(E) has exactly 1 outedge
20b:                        flow ← flow∪Out(Dest(E))
21:             else if E ∈ ssa
22:                 ssa ← ssa − { E }
23:                 if Dest(E) match ⟨ V ← φ(W) ⟩
24:                     VisitPhi(E, Dest(E))
25:                 else if Dest(E) has any executable inedges
26:                     VisitStatement(E, Dest(E))
* 27:   VisitPhi(E_i: edge ∈ CFG(P), S: statement ⟨ V ← φ(W) ⟩ ∈  Dest(E_i)):
28:         for operand w_i ∈ W
29:             if Executable(E_i)
30:                 LatticeCell(E_i , w_i) ← LatticeCell(DefUse(w_i)_Def , w_i)
31:             else
* 32:                LatticeCell(E_i , w_i) ← ⊤
```

33: *VisitStatement*(*E:* edge ∈ *CFG*(*P*), *S:* statement ∈ *Dest*(*E*)):

34: if *S* match ⟨ ⟨ if | while ⟩ *Q* ⟩

35: case *Evaluate*(*E, Q*)

36: ⊥: *flow* ← *flow*∪*Out*(*Dest*(*E*))

37: True: *flow* ← *flow*∪{ $Out(Dest(E))_{\text{True}}$ }

38: False: *flow* ← *flow*∪{ $Out(Dest(E))_{\text{False}}$ }

39: else if *S* match ⟨ *V* ← *W* ⟩

40: *temp* ← *Evaluate*(*E, W*)

41: if *temp* ≠ *LatticeCell*(*E, V*)

42: *LatticeCell*(*E, V*) ← *temp*

43: *ssa* ← *ssa*∪{ edge *e* | *e* ∈ *DefUse*(*V*) }

44: *Evaluate*(*E:* edge ∈ *CFG*(*P*), *D:* expression):

45: if [∃ variable *V* ∈ *D*](*LatticeCell*(*E, V*) = ⊥)

46: return ⊥

47: if *D* match ⟨ $\phi(W)$ ⟩

* 48: return $\prod_{W}$

49: return constant arithmetic on *D* using *LatticeCell*(*E, U*) in place of each variable instance *U*

Algorithm 4. Adapted from specifications by Wegman and Zadeck and lecture notes by Steve Muchnick.

Line 2: Construct the control flow graph in SSA form.

Lines 3–4: Construct a def-use chain for each variable in the program and add all its edges to the control flow graph.

Lines 5–6: Start two worklists: one for control flow edges, and another for SSA edges (def-use edges).

Line 12: The exact wording for this line given in the thesis is, “Halt execution when both worklists become empty. Execution may proceed by processing items from either worklist” (Wegman 192).

Lines 20a and 20b are added by Steve Muchnick in his lecture notes on Wegman and Zadeck, so that their algorithm will be compatible with reducible control flow graphs (Muchnick 6). Wegman and Zadeck write their algorithm in a way that assumes that the control flow graph consists of an irreducible graph of basic blocks of code while the nodes in the def-use graphs consist of individual code statements (Wegman 193). This is confusing. For the purpose of understanding the algorithm, assume that every node contains exactly one code statement (i.e. the construction of the CFG may have been done using basic blocks but was then converted back to one individual statement per node).

Lines 27–32: Edge E_i and operand w_i are subscripted, indicating they are positionally correspondent. E_i is assumed to be the i-th inedge to the ϕ-node, and w_i is assumed to be the i-th operand to the ϕ-function.

Line 32: Apply top element to this cell, because the meet operation is monotonic and will deprioritize the lowest element (which is *top*). Thus, if a particular edge is not executable, its lattice elements will be annihilated in a meet operation.

Line 48: The meet-over-all-paths operation resolves ϕ-functions.

W := a list of arguments

$\prod$:= aggregate form of the meet operation

Analysis

Wegman and Zadeck’s algorithm has no enumerative structures. Instead, the number of times it runs a subprocess depends on the size of two worklists, so the key lines to look out for are any that build up the lists: lines 36–38, and 43. One of the lines in 36–38 is guaranteed to run once for each flow edge

in the CFG, but the execution of Line 43 depends on the evaluation of a lattice element. Thus, it is possible that each each SSA edge could be examined at most twice—once to decline a lattice element from $\top$, and a possible second time to decline from a constant to $\bot$—giving the algorithm a time of $O(2|E_{ssa}| + |E_{flow}|)$, proportional to the size of the complete CFG (Wegman 191, 193).

Recall that Kildall's algorithm (Simple Constant Propagation) runs in $O(|N| \cdot |V|^2)$. Its operative line runs naïvely, performing replacements at every node in the graph, for every variable that's detected to be constant. The Sparse Constant Algorithm leverages this cost with the use of def-use chains, which take a miniscule amount of space in comparison. If a shortcut to all uses of a variable has already been constructed, then the code replacement can be performed just once or twice upon visitation of each edge in the CFG. The advantage of using SSA form in this algorithm comes in its handling of def-use chains, since it guarantees that each variable has a unique definition. Without SSA form, Line 30 above would not be possible, and determining the value at a lattice cell would require the more costly use of reaching definition analysis (Pop 19, Goldstein).

In computer science, one of the bests way to demonstrate the complexity of an algorithm is by showing the graph work involved. As we know by the travelling salesman problem, graph traversals are typically a bad sign when it comes to efficiency, and the best thing a developer can do is find a way to reduce them as much as possible. In the pseudocode example below, the def-use chains for a and b are relatively simple: one vertex and multiple destination vertices; but c is not so lucky. While visually straightforward, the multiplicity of possible values to be checked can very quickly make a program unmanageable. SSA eliminates much of this hassle by making sure that all def-use chains are minimal in their complexity: each having only one source vertex.

Example

Pseudocode	Def-use Chains
1: $a \leftarrow 18$	a : 1 → 2, 8
2: $b \leftarrow 9 - (a / 3)$	b : 2 → 3
3: $c \leftarrow b \cdot 4$	c : 3 → 5, 7
4: if Q	5 → 8
5: $c \leftarrow c - 10$	5 → 8
6: else	
7: $c \leftarrow c + 10$	
8: return $c \cdot (60 / a)$	

Pseudocode in SSA Form	Def-use Chains
1: $a_1 \leftarrow 18$	a_1 : 1 → 2, 9
2: $b_1 \leftarrow 9 - (a_1 / 3)$	b_1 : 2 → 3
3: $c_1 \leftarrow b_1 \cdot 4$	c_1 : 3 → 5, 6
4: if Q	c_2 : 5 → 8
5: $c_2 \leftarrow c_1 - 10$	c_3 : 7 → 8
6: else	c_4 : 8 → 9
7: $c_3 \leftarrow c_1 + 10$	
8: $c_4 \leftarrow \phi(c_2 , c_3)$	
9: return $c_4 \cdot (60 / a_1)$	

Conclusion

Wegman and Zadeck demonstrate the usefulness of SSA form by applying it to a classic problem in code optimization to improve efficiency. Other actors, such as Cytron et al., demonstrate the interest

topic generated in the field of computer science by adding to the theory around it and making incremental improvements to the form.

SSA form makes it possible to perform multiple code optimizations, besides the examples mentioned, concurrently, as well as improving the efficiency of certain established optimization algorithms. By now, SSA form has become one of the basic units in the field of compiler optimization and sees widespread implementation in compiler design, being used in languages like Swift, Erlang, and more notably the LLVM Compiler Infrastructure ("Swift Intermediate Language (SIL)," "Erlang OTP 22.0 is released," Lattner and Adve).

Works Cited

Alpern, Bowen; Wegman, Mark N. and Zadeck, F. Kenneth. "Detecting equality of variables in programs." *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA. 1988. pp. 1–11. doi:10.1145/73560.73561.

Aycock, John and Horspool, Nigel. "Simple generation of static single-assignment form." *Proceedings of the 9th International Conference on Compiler Construction*. 2000. pp. 110–124. doi: 10.1007/3-540-46423-9_8.

Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N. and Zadeck, F. Kenneth. "Efficiently computing static single assignment form and the control dependence graph." *ACM Transactions on Programming Languages and Systems*, 13 (4). 1991. pp. 451–490. CiteSeerX: 10.1.1.100.6361.

Horwitz, Susan B. "Dataflow Analysis" [Online Web page]. 2013. Available: http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html. Accessed: January 2021.

Kildall, Gary A. "A Unified Approach to Global Program Optimization." *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. Boston, Massachusetts, USA. 1 Oct. 1973. pp. 194–206. doi:10.1145/512927.512945. hdl:10945/42162. S2CID 10219496. Archived (PDF) from the original on 29 June 2017. Accessed 20 Nov. 2006.

Lattner, Chris and Adve, Vikram. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." *llvm.org*. University of Illinois at Urbana-Champaign. 30 Jan. 2004. [Online PDF]. Available: https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf. Accessed: 1 April 2021.

McKinley, Kathryn S. "More Data Flow Analysis" [PowerPoint slides]. 2009. Available: https://www.cs.utexas.edu/users/mckinley/380C/lecs/04.pdf. Accessed: January 2021.

Muchnick, Steve. "SSA-based Conditional Constant Propagation." [PowerPoint slides]. 2002. Available: https://cseweb.ucsd.edu/classes/sp02/cse231/lec11seq.pdf. Accessed: January 2021.

Pop, Sebastian. "The SSA Representation Framework: Semantics, Analyses and GCC Implementation." Ph.D. dissertation, Docteur de l'Ecole des Mines de Paris. 13 Dec. 2006. p. 19. Available: http://cri.ensmp.fr/people/pop/papers/2006-12-thesis.pdf. Accessed: 30 March 2021.

Wegman, Mark N. and Zadeck, F. Kenneth. "Constant Propagation with Conditional Branches." *ACM Transactions on Programming Languages and Systems*, 13 (2). April 1991. pages 181–210.

Goldstein, Seth Copen. "Lecture 2: Dataflow Analysis, Basic Blocks, Related Optimizations & SSA" [PowerPoint slides]. 2009. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15745-s09/www/lectures/lect2-df-ssa.pdf. Accessed: 30 March 2021.

"Swift Intermediate Language (SIL)." *github.com*. [Online Web page]. Available: https://github.com/apple/swift/blob/main/docs/SIL.rst. Accessed: 1 April 2021.

"Erlang OTP 22.0 is released." *erlang.com*. 14 May 2019. [Online Web page]. Available: https://www.erlang.org/news/132. Accessed: 1 April 2021.