Static Single Information Form

C. Scott Ananian and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{cananian, rinard}@flexc.lcs.mit.edu

Abstract

This paper presents a new intermediate format called Static Single Information (SSI) form. SSI form generalizes the traditional concept of a variable definition to include all *information definition points*, or points where the analysis may obtain information about the value in a variable. Information definition points include conditional branches as well as assignments. Because SSI form provides a new name for each variable at each information definition point, it provides excellent support for both *predicated analyses*, which exploit information gained from conditionals, and backwards dataflow analyses.

We have developed a Java compiler that uses SSI form as its primary program representation. We have used SSI form to implement several predicated analyses, including redundant array bounds and null pointer check analyses, a conditional constant propagation analysis, and a bit-width analysis. Our experimental results show that the analyses execute efficiently and extract information that can be used to significantly optimize the program. Furthermore, we believe that SSI form significantly simplified the efficient implementation of these analyses.

1 Introduction

Static Single Assignment (SSA) form transforms the program so that exactly one definition of each variable reaches each use. Traditional dataflow analyses, which propagate information from variable definitions to uses, therefore become much simpler to express. Instead of generating an analysis result for each variable at each point in the program, SSA allows the analysis to generate a result for each variable. This sparse representation improves both the simplicity and the efficiency of the analysis.

But definitions are not the only place where an analysis can extract information about the values of variables. Conditional branches also provide information about the values of variables. Consider what happens when one attempts to incorporate this information into an SSA-based analysis. The original problem that SSA eliminated (the need to extract information for each variable at each program point) returns. Variable names do not change at the conditional branch, even though the compiler has different information along the two control-flow paths.

The resulting mismatch between variable names and dataflow information also produces an efficiency problem. Instead of propagating information directly from definitions to uses, the analysis must propagate the information through all program points, whether the program point uses the information or not.

Inspired by this observation, we have developed a new program representation, Static Single Information (SSI) form, that recaptures the advantages of SSA form for predicated analyses, or analyses which use the predicates in conditional branches to extract analysis information. The insight behind this representation is the use of σ-functions, which produce new names for variables at splits in the control flow. The analysis can then associate information with variable names, and propagate the information efficiently and directly from information definition points to uses.

In addition to these practical properties, SSI form has several appealing theoretical properties. It is always possible to place σ-functions so that the number of σ-functions is linear in the size of the original input program. Furthermore, our placement algorithm also runs in linear time. Finally, and perhaps most importantly, SSI form allows us to recast compound dataflow analyses as a flat, unified system of constraints. This formulation allows us to generalize the standard fixed-point solution mechanism for dataflow equations to include constraint resolution rules. The result is a more uniform and powerful analysis framework.

We have implemented a compiler infrastructure for Java, the MIT Flex system, that uses SSI form [7]. We have used this compiler infrastructure to implement several predicate-based analyses, including an analysis that detects redundant array bounds and null reference checks, an analysis that determines the number of bits required to represent values in different variables, and a conditional constant propagation analysis.

Our experimental results show that our analyses execute efficiently and extract information that can be used to significantly optimize the program. Furthermore, we believe that SSI form significantly simplified the implementation of these analyses. Our experiences have led us to use SSI form as the primary program representation in the Flex compiler system.

2 The Static Single Information form

In this section we will provide a formal specification of SSI form and its *minimal* and *pruned* variants. We will also provide efficient algorithms for constructing these representations.

2.1 Definition of SSI form

SSI form is an extension of the SSA form introduced in [4]. Building SSI form involves adding pseudo-assignments for a variable V:

(ϕ) at a control-flow merge when disjoint paths from a conditional branch come together and at least one of the paths contains a definition of V; and

(σ) at locations where control-flow splits and at least one of the disjoint paths from the split uses the value of V.

2.2 Criteria for inserting σ-functions

To minimize the number of σ-functions, there should be a σ-function for variable a at node z of the flowgraph exactly when:

1. node x contains a use of a,
2. node y contains a use of a,
3. there is a nonempty path P_{zx} of edges from z to x,
4. there is a nonempty path P_{zy} of edges from z to y, and
5. paths P_{zx} and P_{zy} do not have any node in common except z (that is, z is the point of divergence for these paths).

We will call this the *path-convergence criterion* for inserting σ-functions. We consider the start node to contain an implicit definition of every variable, and the end node to contain an implicit use of every variable.

Upon examination, we see that the path-convergence criteria for ϕ- and σ-functions interact. Since σ-functions are variable definitions and ϕ-functions are variable uses, the set of equations defined by the respective criteria must be iterated together in order to find the necessary function sets. The total number of ϕ- and σ-functions remains linear, however: we can only place a single ϕ- and/or σ-function per variable at any given flowgraph node, so the total number of added functions is limited to $2 \cdot N \cdot V$.

2.3 Variable renaming after ϕ- and σ-function insertion

Once the compiler has determined where to place the ϕ-functions and σ-functions, it renames variables to satisfy the following two properties:

Property 2.1 (Naming after ϕ-functions.). *For every node x containing a definition of a variable a in the renamed program and node y containing a use of that variable, there exists at least one non-empty path P_{xy} of edges from x to y and no such path contains a definition of a other than at x.*

Property 2.2 (Naming after σ-functions.). *For every pair of nodes x and y containing uses of a variable a defined at a node z in the renamed program, either every nonempty path P_{zx} of edges from z to x must contain node y, or every nonempty path P_{zy} of edges from z to y must contain x.*

In addition, correctness requires that the following hold:

Property 2.3 (Correctness.). *Along any possible control-flow path in a program being executed consider any use of a variable a in the original program and the corresponding use of a_i in the renamed program. Then, at every occurrence of the use on the path, a and a_i have the same value. The path need not be cycle-free.*

2.4 Minimal and pruned SSI forms

Minimal and *pruned* SSI forms can be defined which parallel their SSA counterparts. *Minimal* SSI has the smallest number of ϕ- and σ-functions such that the above conditions are satisfied. *Pruned* SSI form is the minimal form with any unused ϕ- and σ-functions deleted; that is, it contains no ϕ- or σ-functions after which there are no subsequent non-ϕ- or σ-function uses of any of the variables defined on the left-hand side.[1] Figure 1 on the following page compares minimal and pruned SSI form for an example program.

Note that, as in SSA form, pruned SSI does not strictly satisfy the SSI constraints because it omits dead ϕ- and σ-functions otherwise required by the path-convergence criteria. In practice, a subtractive definition of pruned form — generate minimal form and then remove the unused ϕ- and σ-functions — is most useful, but a constructive definition can be generated from the standard SSI form definition as follows:

1. The convergence/divergence node z of the path-convergence criteria for inserting ϕ- and σ-functions must also satisfy: "and there exists a nonempty path P_{zu} from z to a u, a use of a in the original program, which does not contain another definition of a."
2. The boundary condition specified by the path-convergence criterion for the node `END` can be loosened as follows (emphasis indicates modifications): "For the purposes of this definition, the `START` node is assumed to contain a definition for every variable in the original program and the `END` nodes a use *for every variable live at `END`* in the original program."

Pruned form is defined as having the minimal set of ϕ- and σ-functions that satisfy the amended conditions. It can easily be verified that the modifications suffice to eliminate unused ϕ- and σ-functions: if the variable defined in a ϕ- or σ-function is used, there must exist a nonempty path P_{zu} as mandated by amendment 1, where amendment 2 lets $u =$ `END` for variables live exiting the procedure and thus usefully defined.

Property 2.4. *A node Z gets a ϕ- or σ-function for some variable V_i in pruned SSI form only if the corresponding variable V is live at Z in the original program.*

Proof. This is a trivial restatement of amendment 1. A variable v is said to be live at some node N if there exists a node U using v and a path $N \xrightarrow{+} U$ on which no definitions of v are to be found. If V is not live at Z then no path $Z \xrightarrow{+} U$ satisfying the amended path-convergence criteria can be found and neither a ϕ- or σ-function can be placed. Amendment 2 ensures this holds true at boundaries. ☐

3 SSI construction algorithms

Construction of SSI form takes place in two phases. First, the required ϕ- and σ-functions for each variable are inserted at control-flow merge and split points. Then renaming is performed to create a valid SSI form program.

[1] An even more compact SSI form may be produced by removing σ-functions for which there are uses for *exactly one* of the variables on the left-hand side, but by doing so one loses the ability to perform renaming at some control-flow splits which may generate additional value information.

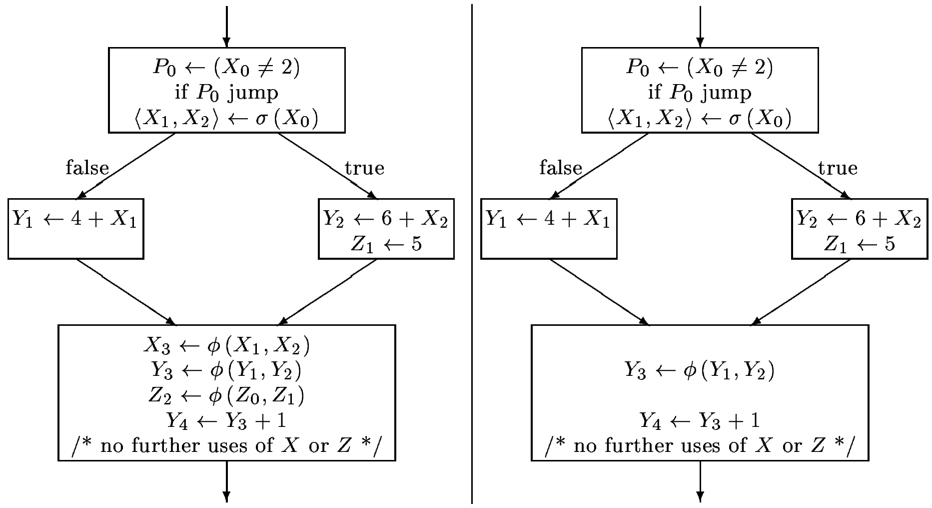

Figure 1: Minimal (left) and pruned (right) SSI forms.

3.1 Placement algorithms

Sreedhar and Gao have shown [18] that it is possible to place ϕ-functions in time proportional to the size of the program. With appropriate modifications to the algorithm, it can be used to place σ-functions. However, as noted above, ϕ- and σ-function placement is not independent: the placement of ϕ-functions necessitates additional σ-function placement, and vice versa. Thus, the (linear time) placement algorithms can be run iteratively to find a fixed point. Since the maximum number of ϕ- or σ-functions is proportional to the size of the program, it is obvious that no more than N iterations will be required, resulting in a worst-case running time of $O(N^2)$. In practice one would expect relatively few iterations[2] yielding a near-linear runtime.

The most common construction algorithm for SSA form [5] uses dominance frontiers and suffers from a possible quadratic blow-up in the size of the dominance frontier for certain common programming constructs. Various improved algorithms use such things as DJ graphs [18] and the dependence flow graph [10] to achieve $O(EV)$ time complexity for ϕ-function placement. We build on this work to achieve $O(EV)$ construction of SSI form, and present a new algorithm for variable renaming in SSI form after ϕ- and σ-functions are placed.

Our construction algorithm begins with a program structure tree of single-entry single-exit (SESE) regions, constructed as described by Johnson, Pearson, and Pingali [9].

We split the construction of SSI form into two parts: placing ϕ- and σ-functions and renaming variables. The placement algorithm runs in $O(NV_0)$ time, and is presented as Algorithm A.1 on page 12. The algorithm is parameterized on a function called `MaybeLive`. For minimal SSI form, `MaybeLive` should always return `true`. Faster practical runtime may be obtained if pruned SSI form is the desired goal by allowing `MaybeLive` to return any conservative approximation of variable liveness information, which will allow early suppression of unused ϕ- and σ-functions. Note that `MaybeLive` need not be precise; conservative values will only result in an excess of ϕ- and σ-functions, not an invalid SSI form. Section 3.1.3 describes a post-processing algorithm to efficiently remove the excess ϕ- and σ-functions.

[2] The number of required iterations is related to the maximum loop nesting depth, which Knuth [13] showed remains small for "human-generated" programs.

Lemma 3.1. *No ϕ-functions (σ-functions) for a variable v are needed in an SESE region not containing a definition (use) of v.*

Proof. See Appendix B. □

Lemma 3.2. *If a definition (use) or a ϕ- or σ-function for a variable v is present at some node D (U), then a ϕ-function (σ-function) for v is needed at every node N:*

1. *of input (output) arity greater than 1,*
2. *reachable from D (from which U is reachable),*
3. *whose smallest enclosing SESE contains D (U), and*
4. *which is not dominated by D (not post-dominated by U).*

Proof. See Appendix B. □

In practice, the conditions of Lemma 3.2 are too expensive to implement directly. Instead, we use a conservative approximation to SSI form, which allows us to place more ϕ- and σ-functions than minimal SSI requires while satisfying the conditions of the SSI form definition. Our algorithm also allows us to do pre-pruning of the SSI form during placement. The result is not pruned SSI, but contains a tight superset of the ϕ- and σ-functions that pruned form requires.

Theorem 3.1. *Algorithm A.1 places all the ϕ- and σ-functions required by the path-convergence criteria for ϕ- and σ-functions.*

Proof. Lemma 3.1 states that the child region exclusion of Algorithm A.1 does not cause required ϕ- or σ-functions to be omitted. Property 2.4 allows the omission of ϕ- and σ-functions for v at nodes where v is dead when creating pruned form; `MaybeLive` may not return `false` for nodes where v is not dead, but may return `true` at nodes where v is dead without harming the correctness of the ϕ- and σ-function placement. □

3.1.1 Computing liveness

Incorporating liveness information into the creation of pruned SSI form appears to lead to a chicken-and-egg problem: although the pruned SSI framework allows highly efficient liveness analysis, obtaining the liveness information from the original program can be problematic. The fastest sparse algorithm has stated time bounds of $O(E + N^2)$ [3], which is likely to be more expensive than the rest of the SSI form conversion. Luckily, Kam and Ullman [11], in conjunction with an empirical study by Knuth [13], show that liveness analysis is highly likely to be linear for reducible flow-graphs. In our work this question is avoided, as we obtain our liveness information directly from properties of the Java bytecode files that are our input to the compiler. But in any case our algorithms allow conservative approximation to liveness, so even in the case of non-reducible flow graphs it should not be difficult to quickly generate a rough approximation.

3.1.2 Variable renaming

We have shown that Algorithm A.1 places all the required ϕ- and σ-functions in the control-flow graph according to the path-convergence criteria for SSI form and the stated boundary conditions at `START` and `END`. The next step is to rename variables to be consistent with properties 2.1 and 2.2. Algorithm A.2 in Appendix A performs this variable renaming. Algorithm A.2 starts on a flow-graph with placed ϕ- and σ-functions. When the algorithm finishes, the control flow-graph will be in proper SSI form. The SSI form is not necessarily minimal. The next section will show how to post-process to create minimal or pruned SSI form.

Theorem 3.2. *Algorithm A.2 renames variables such that SSI form properties 2.1, 2.2, and 2.3 hold.*

Proof. Direct from lemmas B.2, B.3, and B.4. □

Theorem 3.3. *Algorithms A.1 and A.2 correctly transform a program into SSI form.*

Proof. Theorem 3.1 proves that ϕ- and σ-functions are placed correctly to satisfy the path-convergence criteria of the SSI form definition, and theorem 3.2 proves that variables are renamed correctly to satisfy properties 2.1, 2.2 and 2.3. □

3.1.3 Pruning SSI form

The SSI algorithm can be run using any conservative approximation to the liveness information (including the function `MaybeLive`(v, n) = `true`) if unused code elimination[3] is performed to remove extra ϕ- and σ-functions added and create pruned SSI. Figure 17 and Algorithm A.4 present an algorithm to identify unused code in $O(NV_{SSI})$ time, after which a simple $O(N)$ pass suffices to remove it. The complexity analysis is simple: nodes and variables are visited at most once, raising their value in the analysis lattive from *unused* to *used*. Nodes marked *used* are never visted. So `MarkNodeUseful` is invoked at most N times, and `MarkVarUseful` is invoked at most V_{SSI} times. The calls to `MarkNodeUseful` may examine at most every variable use in the program in lines 3-5, taking $O(U_{SSI})$ time at worst. Each call to `MarkVarUseful` examines at most one node (the single definition node for the variable, if it exists) and in constant time pushes at most one node on to the worklist for a total of $O(V_{SSI})$ time. So the total run time of `FindUseful` is $O(U_{SSI} + V_{SSI}) = O(U_{SSI})$.

[3] We follow [19] in distinguishing *unreachable code elimination*, which removes code that can never be executed, from *unused code elimination*, which deletes sections of code whose results are never used. Both are often called "dead code elimination" in the literature.

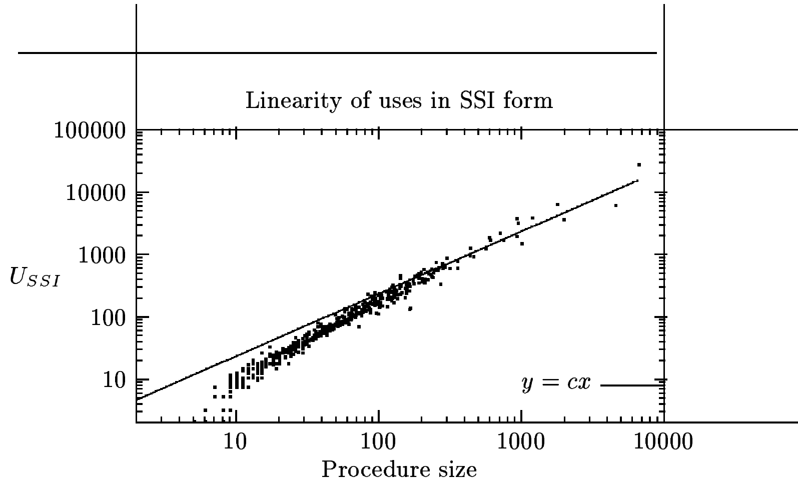

Figure 2: Number of uses in SSI form as a function of procedure length.

3.1.4 Discussion

Note that our algorithm for placing ϕ- and σ-functions in SSI form is *pessimistic*; that is, we at first assume every node in the control-flow graph with input arity larger than one requires a ϕ-function for every variable and every node with out-arity larger than one requires a σ-function for every variable, and then use the PST, liveness information, and unused code elimination to determine safe places to *omit* ϕ- or σ-functions. Most SSA construction algorithms, by contrast, are *optimistic*; they assume no ϕ- or σ-functions are needed and attempt to determine where they are provably necessary. In our experience, optimistic algorithms tend to have poor time bounds because, in the worst case, they may need to perform multiple passes over the graph as they propagate ϕ- or σ-functions. In such cases, a pessimistic algorithm assumes the correct answer at the start, fails to show that any ϕ- or σ-functions can be removed, and terminates in one pass. See Appendix C for more information.

3.2 Time and space complexity of SSI form

Discussions of time and space complexity for sparse evaluation frameworks in the literature are often misleadingly called "linear" regardless of what the O-notation runtime

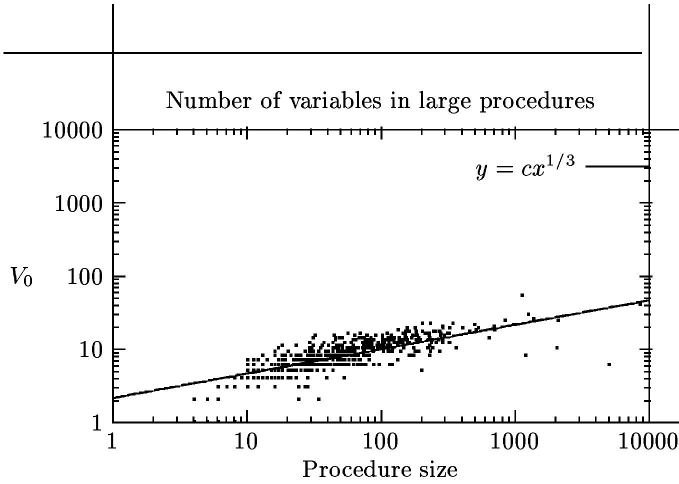

Figure 3: Number of original variables as a function of procedure length.

bounds are. A canonical example is [18], which states that for SSA form, "the number of ϕ-nodes needed remains linear." Typically Cytron [5] is cited; however, that reference actually reads:

> For the programs we tested, the plot in [Figure 21 of Cytron's paper] shows that the number of ϕ-functions is also linear in the size of the original program.

It is important to note that Cytron's claim is based not on algorithmic worst-bounds complexity, but on empirical evidence. This reasoning is not unjustified; Knuth [13] showed in 1974 that "human-generated" programs almost without exception show properties favorable to analysis; in particular shallow maximum loop nesting depth. Wegman and Zadeck [19] clearly make this distinction by noting that:

> In theory the size [of the SSA form representation] can be $O(EV)$, but empirical evidence indicates that the work required to compute the SSA graph is linear in the program size.

Our worst-case space complexity bounds for SSI form are identical to SSA form — $O(EV)$ — but in this section we will endeavour to show that typical complexities are likewise "linear in the program size."

The total runtime for SSI placement and subsequent pruning, including the time to construct the PST, is $O(E + NV_0 + U_{SSI})$. For most programs E will be a small constant factor multiple of N; as Wegman and Zadeck [19] note, most control flow graph nodes will have at most two successors. For those graphs where E is not $O(N)$, it can be argued that E is the more relevant measure of program complexity.

Thus the "linearity" of our SSI construction algorithm rests on the quantities NV_0 and U_{SSI}. Figures 2 and 3 present empirical data for V_0 and U_{SSI} on a sample of 1,048 Java methods. The methods varied in length from 4 to 6,642 statements and were taken from the dynamic call-graph of the FLEX compiler itself, which includes large portions of the standard Java class libraries. Figure 2 shows convincingly that U_{SSI} grows as N for large procedures, and Figure 3 supports an argument that V_0 grows very slowly and that the quantity NV_0 would tend to grow as $N^{1.3}$. This would argue for a near-linear practical run-time.

In contrast, Cytron's original algorithm for SSA form had theoretical complexity $O(E + V_{SSA}|\mathrm{DF}| + NV_{SSA})$. Cytron does not present empirical data for V_{SSA}, but one can infer from the data he presents for "number of introduced ϕ-functions" that V_{SSA} behaves similarly to V_{SSI} — that is, it grows as N, not as V_0. It is frequently pointed out[4] that the $|\mathrm{DF}|$ term, the size of the dominance frontier, can be $O(N^2)$ for common programming constructs (`repeat-until` loops), which indicates that the $V_{SSA}|\mathrm{DF}|$ term in Cytron's algorithm will be $O(N^2)$ at best and at times as bad as $O(N^3)$.

Note that the space complexity of SSI form, which may be $O(EV)$ in the worst case (ϕ- and σ-functions for every variable inserted at every node) is certainly not greater than U_{SSI}, and thus Figure 2 shows linear practical space use.

4 Uses and applications of SSI

The principle benefits of using SSI form are the ability to do predicated and backward dataflow analyses efficiently. ***Predicated analysis*** means that we can use information extracted from branch conditions and control flow. The σ-functions in SSI form provide an variable naming that allows us to sparsely associate the predication information with variable names at control flow splits. The σ-functions also provide a reverse symmetry to SSI form that allow efficient backward dataflow analyses like ***liveness*** and ***anticipatability***.

In this section, we will briefly sketch how SSI form can be applied to backwards dataflow analyses, including anticipatability, an important component of partial redundancy elimination. We will then describe in detail our Sparse Predicated Typed Constant propagation algorithm, which shows how the predication information of SSI form may be used to advantage in practical applications, including the removal of array bounds and null-pointer checks. Lastly, we will describe an extension to SPTC that allows ***bitwidth analysis***, and the possible uses of this information.

4.1 Backward Dataflow Analysis

Backward dataflow analyses are those in which information is propagated in the direction opposite that of program execution [15]. There is general agreement [10, 3, 20] that SSA form is unable to directly handle backwards dataflow analyses; ***liveness*** is often cited as a canonical example.

However, SSI form allows the sparse computation of such backwards properties. Liveness, for example, comes "for free" from pruned SSI form: every variable is live in the region between its use and sole definition. Every non-ϕ-function use of a variable is dominated by the definition; Cytron [5] has shown that ϕ-functions will always be found on the dominance frontier. Thus the live region between definition and use can be enumerated with a simple depth-first search, taking advantage of the topological sorting by dominance that DFS provides [15]. Because of ϕ-function uses, the DFS will have to look one node past its spanning-tree leaves to see the ϕ-functions on the dominance frontier; this does not change the algorithmic complexity.

Computation of other dataflow properties will use this same enumeration routine to propagate values computed

[4] See Dhamdhere [6] for example.

on the sparse SSI graph to the intermediate nodes on the control-flow graph. Formally, we can say that the dataflow property for variable v at node N is dependent only on the properties at nodes D and U, defining and using v, for which there is a path $D \xrightarrow{+} U$ containing N. There is a "default" property which holds for nodes on no such path from a definition to use; for liveness the default property is "not live." The remainder of this section will concentrate on the dataflow properties at use and definition points.

A slightly more complicated backward dataflow property is ***very busy expressions***; this analysis is somewhat obsolete as it serves to save code space, not time. This in turn is related to partial and total ***anticipatability***.

Definition 4.1. *An expression e is* **very busy** *at a point P of the program iff it is always subsequently used before it is killed [15].*

Definition 4.2. *An expression e is* **totally (partially) anticipatable** *at a point P if, on every (some) path in the CFG from P to* `END`*, there is a computation of e before an assignment to any of the variables in e [10].*

Johnson and Pingali [10] show how to reduce these properties of expressions to properties on variables. We will therefore consider properties $\mathrm{BSY}(v,N)$, $\mathrm{ANT}(v,N)$, and $\mathrm{PAN}(v,N)$ denoting very busy, totally anticipatable, and partially anticipatable variables v at some program point N. To compute BSY, we start with pruned SSI form. Any variable defined in a ϕ- or σ-function is used at some point, by definition. So for statements at a point P we have the rules:

$$
\begin{array}{ll}
v = \ldots & \mathrm{BSY}_{\mathrm{in}}(v,P) = \texttt{false} \\
\ldots = v & \mathrm{BSY}_{\mathrm{in}}(v,P) = \texttt{true} \\
x = \phi(y_0,\ldots,y_n) & \mathrm{BSY}_{\mathrm{in}}(y_i,P) = \mathrm{BSY}_{\mathrm{out}}(x,P) \\
\langle x_0,\ldots,x_n\rangle = \sigma(y) & \mathrm{BSY}_{\mathrm{in}}(y,P) = \bigwedge_{i=0}^{n} \mathrm{BSY}_{\mathrm{out}}(x_i,P)
\end{array}
$$

Total anticipatability, in the single variable case, is identical to BSY. Partial anticipatability for a variable v at point P follows the rules:

$$
\begin{array}{ll}
v = \ldots & \mathrm{PAN}_{\mathrm{in}}(v,P) = \texttt{false} \\
\ldots = v & \mathrm{PAN}_{\mathrm{in}}(v,P) = \texttt{true} \\
x = \phi(y_0,\ldots,y_n) & \mathrm{PAN}_{\mathrm{in}}(y_i,P) = \mathrm{PAN}_{\mathrm{out}}(x,P) \\
\langle x_0,\ldots,x_n\rangle = \sigma(y) & \mathrm{PAN}_{\mathrm{in}}(y,P) = \bigvee_{i=0}^{n} \mathrm{PAN}_{\mathrm{out}}(x_i,P)
\end{array}
$$

The present section is concerned more with feasibility than the mechanics of implementation; we refer the interested reader to [15] and [10] for details on how to turn the efficient computation of BSY, PAN and ANT into practical code-hoisting and partial-redundancy elimination routines, respectively.

We note in passing that the sophisticated strength-reduction and code-motion techniques of SSAPRE [12] are applicable to an SSI-based representation, as well, and may benefit from the predication information available in SSI. The remainder of this section will focus on practical implementations of predicated analyses using SSI form.

4.2 Sparse Predicated Typed Constant Propagation

Sparse Predicated Typed Constant (SPTC) Propagation is a powerful analysis tool which derives its efficiency from SSI form. It is built on Wegman and Zadeck's Sparse Conditional Constant (SCC) algorithm [19] and removes unnecessary array-bounds and null-pointer checks, computes variable types, and performs floating-point- and string-constant-propagation in addition to the integer constant propagation of standard SCC.

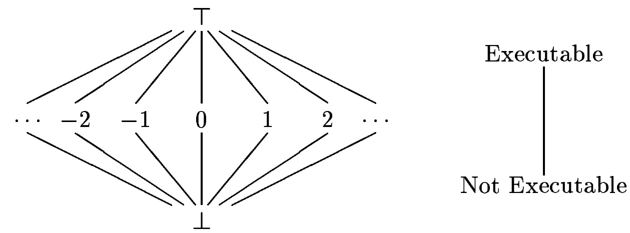

Figure 4: Three-level value lattice and two-level executability lattice for SCC.

$\sqcap$	$\bot$	c	$d(\neq c)$	$\top$
$\bot$	$\bot$	c	d	$\top$
c	c	c	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\oplus$	$\bot$	d	$\top$
$\bot$	$\bot$	d	$\top$
c	c	$c \oplus d$	$\top$
$\top$	$\top$	$\top$	$\top$

Table 1: Meet and binary operation rules on the SCC value lattice.

We will describe this algorithm incrementally, beginning with the standard SCC constant-propagation algorithm. Wegman and Zadeck's algorithm operates on a program in SSA form; we will call this SCC/SSA to differentiate it from SCC/SSI, which uses the SSI form. Section 6 on page 10 will discuss an extension to SPTC which does ***bit-width analysis***.

4.2.1 Wegman and Zadeck's SCC/SSA algorithm

The SCC algorithm works on a simple three-level value lattice associated with variable definition points and a two-level executability lattice associated with flow-graph edges. These lattices are shown in Figure 4. The SCC algorithm itself, which runs in $O(E + U_{SSA})$ time, is presented in Figures A.5 and A.6 from Appendix A.

4.2.2 SCC/SSI: predication using σ-functions.

Porting the SCC algorithm from SSA to SSI form (so that it takes information from conditionals into account) immediately increases the number of constants we can find. Only the `Visit` procedure must be updated for SCC/SSI: lattice update rules for σ-functions must be added. Algorithm 4.1 shows a new `Visit` procedure for the two-level integer constant lattice of Wegman and Zadeck's SCC/SSA; with this restricted value set only integer equality tests tap the algorithm's full power. The utility of SCC/SSI's ***predicated analysis*** will become more evident as the value lattice is extended to cover more constant types. The time complexity of the updated algorithm is identical to that of SCC/SSA: $O(E + U_{SSA})$.

```
Visit(n:node) =
  /* Assignment rules as on page 14 */

  for each branch "if x = y goto e1 else e2" in n do
    if L[x] = ⊤ or L[y] = ⊤ then
      RaiseE(e1)
      RaiseE(e2)
    else if L[x] = c and L[y] = d then
      if c = d then
        RaiseE(e1)
      else
        RaiseE(e2)
    for each assignment "⟨v1, v2⟩ ← σ(v0)" associated with this branch do
      if edge e1 ∈ Ee and variable v0 is the x or y in the test then
        RaiseV(v1, min(L[x], L[y]))
      else if edge e1 ∈ Ee then
        RaiseV(v1, L[v0])
      if edge e2 ∈ Ee then /* False branch */
        RaiseV(v2, L[v0])

  /* Obvious generalization applies for tests like "x ≠ y" */
```

Algorithm 4.1: A revised `Visit` procedure for SCC/SSI.

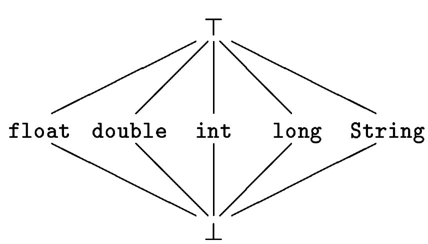

Figure 5: SCC value lattice extended to Java primitive value domain.

4.2.3 Extending the value domain

The first simple extension of the SCC value lattice enables us to represent floating-point and other values. For this work, we extended the domain to cover the full type system of Java bytecode [8]; the extended lattice is presented in Figure 5. The figure also introduces the abbreviated lattice notation we will use through the following sections; it is understood that the lattice entry labelled "int" stands for a finite-but-large set of incomparable lattice elements, consisting (in this case) of the members of the Java `int` integer type. Java `int`s are 32 bits long, so the "int" entry abbreviates 2^{32} lattice elements. Similarly, the "double" entry encodes not the infinite domain of real numbers, but the domain spanned by the Java `double` type which has fewer than 2^{64} members.[5] The Java `String` type is also included, to allow simple constant string coalescing to be performed. The propagation algorithm over this lattice is a trivial modification to Algorithm 4.1, and will be omitted for brevity. In the next sections, the "int" and "long" entries in this lattice will be summarized as "Integer Constant", the "float" and "double" entries as "Floating-point Constant", and the "String" entry as "String Constant". As the lattice is still only three levels deep, the asymptotic runtime complexity is identical to that of the previous algorithm.

4.2.4 Type analysis

In Figure 6 we extend the lattice to compute Java type information. The new lattice entry marked "Typed" is actually forest-structured as shown in Figure 7; it is as deep as the class hierarchy, and the roots and leaves are all comparable to $\top$ and $\perp$. Only the `Visit` procedure must be modified; the new procedure is given as Algorithm 4.2. Because the lattice L is deeper, the asymptotic runtime complexity is now $O(E + U_{SSA}D_c)$ where D_c is the maximum depth of the class hierarchy. To form an estimate of the magnitude of D_c, Table 2 compares class hierarchy statistics for several large object-oriented projects in various source languages. Our FLEX compiler infrastructure, as a typical Java example, has an average class depth of 1.91.[6] In a forced example, of course, one can make the class depth $O(N)$; however, one can infer from the data given that in real code the D_c term is not likely to make the algorithm significantly non-linear.

A brief word on the roots of the hierarchy forest in Figure 7 is called for: Java has both a class hierarchy, rooted at `java.lang.Object`, and several primitive types, which we will also use as roots. The primitive types include `int`, `long`, `float`, and `double`.[7] Integer constants in the lattice are comparable to and less than the `int` or `long` type; floating-point constants are likewise comparable to and less than either `float` or `double`. String constants are comparable to and less than the `java.lang.String` non-primitive class type.

The `void` type, which is the type of the expression `null`, is also a primitive type in Java; however we wish to keep $x \sqcap y$ identical to $\bigsqcup_L \{x, y\}$ (the least upper bound of x and y) while satisfying the Java typing rule that $\texttt{null} \sqcap x = x$ when x is a non-primitive type and not a constant. This

[5] In IEEE-standard floating-point, some possible bit patterns are not valid number encodings.

[6] Measured August 2, 1999; the infrastructure is under continuing development.

[7] In the type system our infrastructure uses (which is borrowed from Java bytecode) the `char`, `boolean`, `short` and `byte` types are folded into `int`.

Hierarchy	Source language	Classes	Avg. depth	Max. depth
FLEX infrastructure	Java	550	1.9	5
`javac` compiler	Java	304	2.8	7
NeXTStep 3.2†	Objective-C	488	3.5	8
Objectworks 4.1†	Smalltalk	774	4.4	10

† indicates data obtained from Muthukrishnan and Müller [14].

Table 2: Class hierarchy statistics for several large O-O projects.

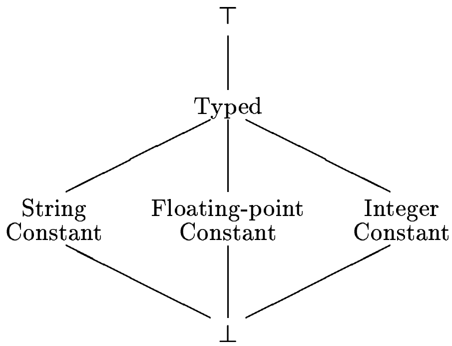

Figure 6: SCC value lattice extended with type information.

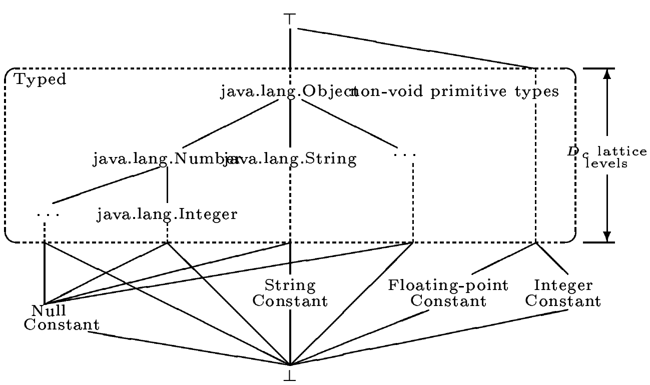

Figure 7: "Typed" category of Figure 6 shown expanded.

$$
\begin{aligned}
\texttt{int} \oplus \texttt{int} &= \texttt{int} \\
\texttt{long} \oplus \{\texttt{int}, \texttt{long}\} &= \texttt{long} \\
\texttt{float} \oplus \{\texttt{int}, \texttt{long}, \texttt{float}\} &= \texttt{float} \\
\texttt{double} \oplus \{\texttt{int}, \texttt{long}, \texttt{float}, \texttt{double}\} &= \texttt{double} \\
\texttt{String} \oplus \{\texttt{int}, \texttt{long}, \texttt{float}, \texttt{double}, \texttt{Object}, \ldots\} &= \texttt{String}
\end{aligned}
$$

Figure 8: Java typing rules for binary operations.

Visit(n:node) =
- **for each** assignment "$v \leftarrow x \oplus y$" in n **do**
  - RaiseV(v, $V[x] \oplus V[y]$) /* *binop rule: see figure 8* */
- **for each** assignment "$v \leftarrow$ MEM($\ldots$)" or "$v \leftarrow$ CALL($\ldots$)" in n **do**
  - **let** t be the type of the `MEM` or `CALL` expression
  - RaiseV(v, t)
- **for each** assignment "$v \leftarrow \phi(x_1, \ldots, x_n)$" in n **do**
  - **for each** variable x_i corresponding to predecessor edge e_i of n **do**
    - **if** $e_i \in E_e$ **then**
      - RaiseV(v, $\bigsqcup_L \{V[v], V[x_i]\}$) /* *meet rule: use least upper bound* */
- **for each** branch "if $x = y$ goto e_1 else e_2" in n **do**
  - **if** Typed $\sqsubseteq L[x]$ or Typed $\sqsubseteq L[y]$ **then**
    - RaiseE(e_1)
    - RaiseE(e_2)
  - **else if** $L[x] = c$ and $L[y] = d$ **then** /* *if x and y are constants...* */
    - **if** $c = d$ **then**
      - RaiseE(e_1)
    - **else**
      - RaiseE(e_2)
  - **for each** assignment "$\langle v_1, v_2 \rangle \leftarrow \sigma(v_0)$" associated with this branch **do**
    - **if** edge $e_1 \in E_e$ and variable v_0 is the x or y in the test **then**
      - /* *type error in source program if $L[x]$ and $L[y]$ are incomparable* */
      - RaiseV(v_1, min($L[x]$, $L[y]$))
    - **else if** edge $e_1 \in E_e$ **then**
      - RaiseV(v_1, $L[v_0]$)
    - **if** edge $e_2 \in E_e$ **then** /* *False branch* */
      - RaiseV(v_2, $L[v_0]$)

/* *Obvious generalization applies for tests like "$x \neq y$"* */
/* *Obvious generalization applies for tests like "x instanceof C"* */

Algorithm 4.2: `Visit` procedure for typed SCC/SSI.

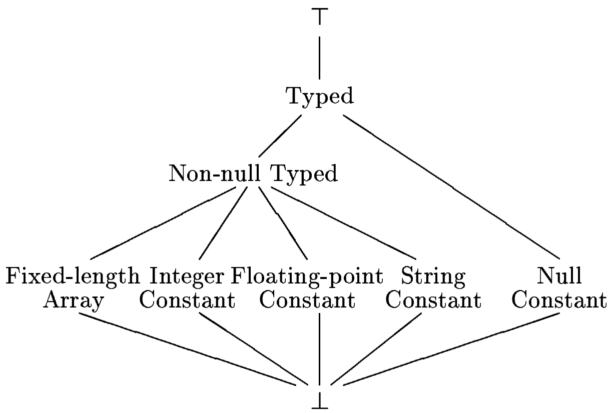

Figure 9: Value lattice extended with array and null information.

$$\forall C \in \text{Class},\ C_{\text{non-null}} \sqsubset C_{\text{possibly-null}}$$
$$\forall C \in \text{Class}_{\text{non-null}},\ \bigsqcup_L \{\texttt{void}, C\} \in \text{Class}_{\text{possibly-null}}$$
$$\forall C \in \text{Class}_{\text{possibly-null}},\ \texttt{void} \sqsubset C$$
$$\forall C \in \text{Class}_{\text{non-null}},\ \langle \texttt{void}, C\rangle \notin \sqsubseteq$$

Let $A(C, n)$ be a function to turn a lattice entry representing a non-null array class type C into the lattice entry representing a said array class with known integer constant length n. Then for any non-null array class C and integers i and j,

$$A(C, i) \sqsubset C$$
$$\langle A(C, i), A(C, j)\rangle \in \sqsubseteq \ \text{ if and only if } i = j$$

Figure 10: Extended value lattice inequalities.

Visit(n:node) =

/* *Binop and* ϕ*-function rules as in algorithm 4.2* */

for each assignment "$v \leftarrow$ MEM(...)" or "$v \leftarrow$ CALL(...)" in n **do**
  let $t \in \text{Class}_{\text{possibly-null}} \cup \text{Class}_{\text{primitive}}$ be the type of the `MEM` or `CALL`
  RaiseV(v, t)

for each array creation expression "$v \leftarrow$ new T[x]" **do**
  if $L[x]$ is an integer constant **then**
    RaiseV(v, $A(T, L[x])$)
  else
    RaiseV(v, $T_{\text{non-null}}$)

for each array length assignment "$v \leftarrow$ arraylength(x)" **do**
  if $L[x]$ is an array of known constant length n **then**
    RaiseV(v, n)
  else
    RaiseV(v, `int`)

/* *Branch rules as in algorithm 4.2, with the obvious extension to allow tests against* `null` *to lower a lattice value from* $\text{Class}_{\text{possibly-null}}$ *to* $\text{Class}_{\text{non-null}}$. */

Algorithm 4.3: `Visit` procedure outline with array and null information.

```
x = 5 + 6;
do {
    y = new int[x];
    z = x-1;
    if (0 <= z && z < y.length)
        y[z] = 0;
    else
        x--;
} while (P);
```

Figure 11: An example illustrating the power of combined analysis.

requires putting `void` comparable to but less than every non-primitive leaf in the class hierarchy lattice.

The Java class hierarchy also includes ***interfaces***, which are the means by which Java implements multiple inheritance. Base interface classes (which do not extend other interfaces) are additional roots in the hierarchy forest, although no examples of this are shown in Figure 7.

Since untypeable variables are generally forbidden, no operation should ever raise a lattice value above "Typed" to $\top$. The otherwise-unnecessary $\top$ element is retained to indicate error conditions.

This variant of the constant-propagation algorithm allows us to eliminate unnecessary `instanceof` checks due to type-casting or type-safety checks. Section 5 will provide experimental validation of its utility.

Finally, note that the ability to represent `null` as the `void` type in the lattice begins to allow us to address null-pointer checks, although because $\texttt{null} \sqcap x = x$ for non-primitive types we can only reason about variables which can be proven to be null, not those which might be proven to be non-null (which is the more useful case). The next section will provide a more satisfactory treatment.

4.2.5 Array-bounds and null-pointer checks

At this point, we can expand the value lattice once more to allow elimination of unnecessary array-bounds and null-pointer checks, based on our constant-propagation algorithm. The new lattice is shown in Figure 9; we have split the "Typed" lattice entry to enable the algorithm to distinguish between non-null and possibly-null values,[8] and added a lattice level for arrays of known constant length. Some formal definition of the new value lattice can be found in Figure 10; the meet rule is still the least upper bound on the lattice. Modifications to the `Visit` procedure are outlined in Algorithm 4.3. Notice that we exploit the pre-existing integer-constant propagation to identify constant-length arrays, and that our integrated approach allows one-pass optimization of the program in Figure 11.

Note that the variable renaming performed by the SSI form at control-flow splits is essential in allowing the algo-

[8] Values which are always-null were discussed in the previous section; they are identified as having primitive type `void`.

```
if (10 < 0)
    throw new NegativeArraySizeException();
int[] A = new int[10];
if (0 < 0 || 0 >= A.length)
    throw new ArrayIndexOutOfBoundsException();
A[0] = 1;
for (int i=1; i < 10; i++) {
    if (i < 0 || i >= A.length)
        throw new ArrayIndexOutOfBoundsException();
    A[i] = 0;
}
```

Figure 12: Implicit bounds checks (underlined) on Java array references.

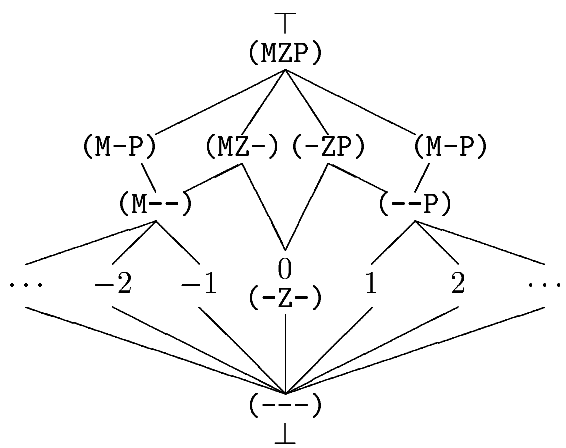

Figure 13: An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain. The (M-P) entry is duplicated to aid clarity.

rithm to do null-pointer check elimination. However, the lattice we are using can remove bound checks from an expression $A[k]$ when k is a constant, but not when k is an bounded induction variable. In the example of Figure 12, the first two implicit checks are optimized away by this version of the algorithm, but the loop-borne test is not.

A typical array-bounds check (as shown in the example on the current page) verifies that the index i of the array reference satisfies the condition $0 \leq i < n$, where n is the length of the array.[9] By identifying integer constants as either positive, negative, or zero the first half of the bounds check may be eliminated. This requires a simple extension of the integer constant portion of the lattice, outlined in Figure 13, with negligible performance cost. However, handling upper bounds completely requires a symbolic analysis that is out of the current scope of this work. Future work will use induction variable analysis and integrate an existing linear programming approach [17] to fully address array-bounds checks.

[9]Languages in which array indices start at 1 can be handled by slight modifications to the same techniques.

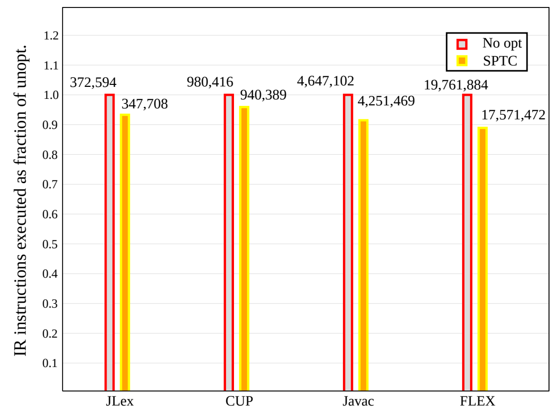

Figure 14: SPTC optimization performance.

5 Experimental results

The full SPTC analysis and optimization has been implemented in the FLEX Java compiler platform. Some quantitative measure of the utility of SPTC is given as Figure 14. The run-times are intermediate representation dynamic statement counts generated by the FLEX compiler SSI IR interpreter. The standard Wegman-Zadeck SCC algorithm, which has proven utility in practice, shows no improvement over unoptimized code due to the metric used. Even so, SPTC shows a 10% speed-up. It is expected that the improvement given in actual practice will be greater.

Note that the speed-up is constant despite widely differing test cases. Even a simple example actually executes quite a bit of library code in the Java implementation; this includes numerous element-by-element array initializations (due to the semantics of java bytecode) which we expect SPTC to excel at optimizing. But SPTC does just as well on the full FLEX compiler (68,032 lines of source at the time the benchmark was run), which shows that the speed-up is not limited to constant initialization code.

6 Bit-width analysis

The SPTC algorithm can be extended to allow efficient ***bit-width analysis***. Bit-width analysis is a variation of constant propagation with the goal of determining value ranges for variables. In this sense it is similar to, but simpler than, array-bounds analysis: no symbolic manipulation is required and the value lattice has N levels (where N is the maximum bitwidth of the underlying datatype) instead of 2^N. For C and Java programs, this means that only 32 levels need be added to the lattice; thus the bit-width analysis can be made efficient.

Bit-width analysis allows optimization for modern media-processing instruction set extensions which typically offer vector processing of limited-width types. Intel's MMX extensions, for example, offer packed 8-bit, 16-bit, 32-bit and 64-bit vectors [16]. To take advantage of these functional units without explicit human annotation, the compiler must be able to guarantee that the data in a vector can be expressed using the limited bit-width available. A simpler bit-width analysis in a previous work [1] showed that

$$
\begin{aligned}
-\langle M,P\rangle &= \langle P,M\rangle \\
\langle M_l,P_l\rangle + \langle M_r,P_r\rangle &= \langle 1+\max(M_l,M_r), 1+\max(P_l,P_r)\rangle \\
\langle M_l,P_l\rangle \times \langle M_r,P_r\rangle &= \langle \max(M_l+P_r, P_l+M_r), \max(M_l+M_r, P_l+P_r)\rangle \\
\langle 0,P_l\rangle \wedge \langle 0,P_r\rangle &= \langle 0, \min(P_l,P_r)\rangle \\
\langle M_l,P_l\rangle \wedge \langle M_r,P_r\rangle &= \langle \max(M_l,M_r), \max(P_l,P_r)\rangle
\end{aligned}
$$

Figure 15: Some combination rules for bit-width analysis.

a large amount of width-limit information can be extracted from appropriate source programs; however, that work was not able to intelligently compute widths of loop-bound variables due to the limitations of the SSA form. Extending the bitwidth algorithm to SSI form allows induction variables width-limited by loop-bounds to be detected.

Bit-width analysis is also a vital step in compiling a high-level language to a hardware description. General purpose programming languages do not contain the fine-grained bit-width information that a hardware implementation can take advantage of, so the compiler must extract it itself. The work cited showed that this is viable and efficient.

The bit-width analysis algorithm has been implemented in the FLEX compiler infrastructure. Because most types in Java are signed, it is necessary to separate bit-width information into "positive width" and "negative width." This is just an extension of the signed value lattice of Figure 13 to variable bit-widths. In practice the bit-widths are represented by a tuple, extending the integer constant lattice with $(\mathit{Int} \times \mathit{Int})_\perp$ under the natural total ordering of *Int*. The tuple $\langle 0,0\rangle$ is identical to the constant 0, and the tuple $\langle 0,16\rangle$ represents an ordinary unsigned 16-bit data type. The $\top$ element is represented by an appropriate tuple reflecting the source-language semantics of the value's type. Figure 15 presents bit-width combination rules for some unary negation and binary addition, multiplication and bitwise-and. In practice, the rules would be extended to more precisely handle operands of zero, one, and other small constants.

7 Conclusion

This paper presents a new intermediate format, Static Single Information (SSI) form. In addition to traditional variable definition points, SSI form provides new names for each variable at each point where the analysis may obtain information about the value in the variable. The form provides excellent support for predicated analyses, which use the information present in conditional branches, because it enables the analyses to propagate information directly from information definition sites to information use sites.

We have implemented a Java compiler that uses SSI form as its primary intermediate representation, and implemented a variety of analyses using the form. These analyses include redundant array bounds and null pointer check analyses, a conditional constant propagation analysis, and a bit-width analysis. Our experimental results show that the analyses execute efficiently and extract information that can be used to significantly optimize the program. Furthermore, we believe that SSI form significantly simplified the efficient implementation of these analyses.

References

[1] C. Scott Ananian. Silicon C: A hardware backend for SUIF. Available from `http://flex-compiler.lcs.mit.edu/SiliconC/paper.pdf`, May 1998.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages (POPL)*, pages 55–66, Orlando, Florida, January 1991.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL)*, pages 25–35, Austin, Texas, January 1989.

[5] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, San Francisco, California, June 1992.

[7] The FLEX compiler project. `http://flex-compiler.lcs.mit.edu/`.

[8] James Gosling. Java intermediate bytecodes. In *Papers of the First ACM SIGPLAN workshop on Intermediate Representations*, pages 111–118, San Francisco, California, January 1995.

[9] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185, Orlando, Florida, June 1994.

[10] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–89, Albuquerque, New Mexico, June 1993.

[11] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 21(3):158–171, January 1976.

[12] Robert Kennedy, Fred Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via SSAPRE. In *Proceedings of the Seventh International Conference on Compiler Construction*, pages 144–158, Lisbon, Portugal, April 1998.

[13] Donald Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(12):105–134, 1974.

[14] S. Muthukrishnan and Martin Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, Atlanta, Georgia, January 1996.

[15] Carl D. Offner. Notes on graph algorithms used in optimizing compilers. Available from `http://www.cs.umb.edu/~offner/files/flow_graph.ps`, 1995.

[16] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, January 1997.

[17] R. Rugina and M. Rinard. Symbolic analysis of divide and conquer programs. In *Submitted to PLDI '00*.

[18] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 62–73, San Francisco, California, January 1995.

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[20] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–310, Portland, Oregon, January 1994.

A Algorithms

B Proofs

Proof of Lemma 3.1.

Proof. Let us assume a ϕ-function for v is needed at some node Z inside an SESE not containing a definition of v. Then by the path-convergence criterion for ϕ-functions, there exist paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ having no nodes but Z in common where X and Y contain either definitions of v or ϕ- or σ-functions for v. Choose any such paths:

Case I: Both X and Y are outside the SESE. Then, as there is only one entrance edge into the SESE, the paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ must contain some node in common other than Z. But this contradicts our choice of X and Y.

Case II: At least one of X and Y must be inside the SESE. If both X and Y are not definitions of v but rather ϕ- or σ-functions for v, then by recursive application of this proof there must exist some choice of X, Y, and Z inside this SESE where at least one of X and Y is a definition. But X or Y cannot be a definition of v because they are inside the SESE of Z which was chosen to contain no definitions of v.

```
Place(G: CFG) =
  let r be the top-level region for G
  for each variable v in G do
    PlaceOne(r, v, false) /* place phis */
    PlaceOne(r, v, true) /* place sigmas */

PlaceOne(r: region, v: variable, ps: boolean): boolean =
  /* Post-order traversal */
  flag ← false
  for each child region r′ do
    if PlaceOne(r′, v, ps) then
      flag ← true

  for each node n in region r not contained in a child region do
    if ps is false and n contains a definition of v then
      flag ← true
    if ps is true and n contains a use of v then
      flag ← true

  /* add phis/sigmas to merges/splits where v may be live */
  if flag = true then
    for each node n in region r not contained in a child region do
      if MaybeLive(v, n) = true then
        if ps is false and the input arity of n exceeds 1 then
          place a phi function for v at n
        if ps is true and the output arity of n exceeds 1 then
          place a sigma function for v at n

  return flag
```

Algorithm A.1: Placing ϕ- and σ-functions.

Data type `Environment`:

create(): Environment :
make an environment with no mappings.

put($\mathcal{E}$: Environment, v_1: variable, v_2: variable) :
extend environment $\mathcal{E}$ with a mapping from v_1 to v_2.

get($\mathcal{E}$: Environment, v: variable): variable :
return the current mapping in $\mathcal{E}$ for v.

beginScope($\mathcal{E}$: Environment) :
save the current mapping of $\mathcal{E}$ for later restoration.

endScope($\mathcal{E}$: Environment) :
restore the mapping of $\mathcal{E}$ to that present at the last `beginScope` on $\mathcal{E}$.

Figure 16: Environment datatype for the SSI renaming algorithm.

```
Rename(G: CFG) =
  Init(G)
  for each edge e leaving START do
     Search(e)

Init(G: CFG) =
  for each edge e in G do
     Marked[e] ← false
  for each variable V in G do
     C(V) ← 0
  ℰ = create() /* create a new environment */

Inc(ℰ: Environment, V: variable): variable =
  i ← C(V) + 1
  C(v) ← i
  ℰ.put(V, V_i)
  return V_i
```

Algorithm A.2: SSI renaming algorithm.

```
Search(⟨s, d⟩: edge) =
Require: s to be a node containing φ- or σ-functions, or START
Require: Marked[⟨s, d⟩] = false
  Marked[⟨s, d⟩] ← true
  beginScope(ℰ)
  if s is a node containing φ-functions then
     for each φ-function P in s do
        replace the destination V of P by Inc(ℰ, V)
  else if s is a node containing σ-functions then
     for each σ-function S in s do
        j ← WhichSucc(⟨s, d⟩)
        replace the j-th destination V of S by Inc(ℰ, V)
  loop /* now rename inside basic block */
     if d is a node containing φ-functions then
        for each φ-function P in d do
           j ← WhichPred(⟨s, d⟩)
           replace the j-th operand V of P by get(ℰ, V)
        break /* end of basic block */
     else if s is a node containing σ-functions then
        for each σ-function S in d do
           replace the operand V of S by get(ℰ, V)
        break /* end of basic block */
     /* ordinary assignment, at most one successor */
     for each variable V in RHS(d) do
        replace V by get(ℰ, V) in RHS(d)
     for each variable V in LHS(d) do
        replace V by Inc(ℰ, V) in LHS(d)
     if d has no successor then
        break /* end of basic block */
     s ← d
     d ← successor of d
  end loop
  for each successor n of d do
     if not Marked[⟨d, n⟩] then
        Search(⟨d, n⟩) /* dfs recursion */
  endScope(ℰ)
  return
```

Algorithm A.3: SSI renaming algorithm, cont.

Operations on nodes:

NodeUseful(n:node): boolean : Whether the results of this node are ever used

Uses(n:node): set of variables : Variables for which this node contains a use

Operations on variables:

VarUseful(v:variable): boolean : Whether there is some **n** for which **Uses(n)** contains **v** and **NodeUseful(n)** is `true`

Definitions(v:variable): set of nodes : Nodes which contain a definition for **v**

Figure 17: Datatypes and operations used in unused code elimination.

```
FindUseful(G: CFG) =
  let W be an empty work list
  for each variable v in G do
     VarUseful(v) ← false
  for each node n in G in any order do
     NodeUseful(n) ← false
     if n is a CALL, RETURN, or other node with side-effects then
        add n to W

  while W is not empty do
     let n be any element from W
     remove n from W
     MarkNodeUseful(n, W)

MarkNodeUseful(n: node, W: WorkList) =
  NodeUseful(n) ← true
  /* everything used by a useful node is useful */
  for each variable v in Uses(n) do
     if not VarUseful(v) then
        MarkVarUseful(v, W)

MarkVarUseful(v: variable, W: WorkList) =
  VarUseful(v) ← true
  /* The definition of a useful variable is useful */
  for each node n in Definitions(v) do
     /* In SSI form, size(Definitions(v)) ≤ 1 */
     if not NodeUseful(n) then
        add n to W
```

Algorithm A.4: Identifying unused code using SSI form.

```
Init(G:CFG) =
  E_e ← ∅
  E_n ← ∅
  for each variable v in G do
    if some node n defines v then
      V[v] ← ⊥
    else
      V[v] ← ⊤ /* Procedure arguments, etc. */

Analyze(G:CFG) =
  let r be the start node of graph G
  E_n ← E_n ∪ {r}
  W_n ← {r}
  W_v ← ∅

  repeat
    if W_n is not empty then
      remove some node n from W_n
      if n has only one outgoing edge e and e ∉ E_e then
        RaiseE(e)
      Visit(n)
    if W_v is not empty then
      remove some variable v from W_v
      for each node n containing a use of v do
        Visit(n)
  until both W_v and W_n are empty
```

Algorithm A.5: SCC algorithm for SSA form.

```
RaiseE(e:edge) =
  /* When called, e ∉ E_e */
  E_e ← E_e ∪ {e}
  let n be the destination of edge e
  if n ∉ E_n then
    E_n ← E_n ∪ {n}
    W_n ← W_n ∪ {n}

RaiseV(v:variable, L:lattice value) =
  if V[v] ⊏ L then
    V[v] ← L
    W_v ← W_v ∪ {v}

Visit(n:node) =
  for each assignment "v ← x ⊕ y" in n do
    RaiseV(v, V[x] ⊕ V[y]) /* binop rule: see table 1 */

  for each assignment "v ← MEM(...)" or "v ← CALL(...)"
  in n do
    RaiseV(v, ⊤)

  for each assignment "v ← φ(x_1, ..., x_n)" in n do
    for each variable x_i corresponding to predecessor edge e_i
    of n do
      if e_i ∈ E_e then
        RaiseV(v, V[v] ⊓ V[x_i]) /* meet rule: see table 1 */

  for each branch "if v goto e_1 else e_2" in n do
    L ← V[v]
    if L = ⊤ or L = c where c signifies "true" and e_1 ∉ E_e
    then
      RaiseE(e_1)
    if L = ⊤ or L = c where c signifies "false" and e_2 ∉ E_e
    then
      RaiseE(e_2)
```

Algorithm A.6: SCC algorithm for SSA form, cont.

A symmetric argument holds for σ-functions for v, using the path-convergence criterion for σ-functions, and the fact that there exists one exit edge from the SESE. □

Proof of Lemma 3.2.

Proof. We will first prove that a node N failing any one of the conditions does not need a ϕ- or σ-function.

- The path-convergence criteria for ϕ-functions (σ-functions) require node N to be the first convergence (divergence) of some paths $X \stackrel{+}{\to} N$ and $Y \stackrel{+}{\to} N$ ($N \stackrel{+}{\to} X$ and $N \stackrel{+}{\to} Y$). If the input arity is less than 2 or there is no path from a definition of v, than it fails the path-convergence criterion for ϕ-functions. If the output arity is less than 2 or there is no path to a use of v, then it fails the path-convergence criterion for σ-functions.
- If there exists a SESE containing N that does not contain any definition, ϕ- or σ-function D for v, then N does not require a ϕ- or σ-function for v by lemma 3.1.
- Let us suppose every D_i containing a definition, ϕ- or σ-function for v dominates N. If N requires a ϕ-function for v, there exist paths $D_1 \stackrel{+}{\to} N$ and $D_2 \stackrel{+}{\to} N$ containing no nodes in common but N. We use these paths to construct simple paths $\texttt{START} \stackrel{+}{\to} D_1 \stackrel{+}{\to} N$ and $\texttt{START} \stackrel{+}{\to} D_2 \stackrel{+}{\to} N$. By the definition of a dominator, every path from `START` to N must contain every D_i. But $D_1 \stackrel{+}{\to} N$ cannot contain D_2, and if $\texttt{START} \stackrel{+}{\to} D_1$ contains D_2, we can make a path $\texttt{START} \stackrel{+}{\to} D_2 \stackrel{+}{\to} N$ which does not contain D_1 by using the D_1-free path $D_2 \stackrel{+}{\to} N$. The assumption leads to a contradiction; thus, there must exist some D_i which does not dominate N if N is required to have a ϕ-function for v. The symmetric argument holds for post-dominance and σ-functions.

This proves that the conditions are necessary. It is obvious from an examination of the path convergence criteria for ϕ- and σ-functions and lemma 3.1 that they are sufficient. □

The SSI renaming algorithm presented in Figures A.2 and A.3 requires an `Environment` datatype which is defined in Figure 16. Using an imperative programming style, it is possible to perform a sequence of any N operations on `Environment` as defined in the figure in $O(N)$ time; in a functional programming style any N operations can be completed in $O(N \log N)$ time.[10] As the coarse structure of Algorithm A.2 is a simple depth-first search, it is easy to see that the `Search` procedure can be invoked from line 3 on page 13 and line 32 on page 13 a total of $O(E)$ times; likewise its inner loop (lines 10 to 29) can be executed a total of E times across all invocations of `Search`. A total of $U_{SSA} + D_{SSA}$ calls to the operations of the `Environment` datatype will be made within all executions of `Search`. For the imperative implementation of `Environment` a total time bounds of $O(E + U_{SSA} + D_{SSA})$ for the variable renaming algorithm is obtained.

[10] The curious reader is referred to section 5.1 of Appel [2] for implementation details.

Lemma B.1. *The stack trace of calls to* `Search` *defines a unique path through* G *from* *START*.

Proof. We will prove this lemma by construction. For every consecutive pair of calls to `Search` we construct a path $X \xrightarrow{+} Y$ starting with the edge $\langle X, N_0\rangle$ which is the argument of the first call, and ending with the edge $\langle N_n, Y\rangle$ which is the argument of the second call. From line 28 of the `Search` procedure on page 13 we note that every edge $\langle N_i, N_{i+1}\rangle$ between the first and last has exactly one successor. Furthermore, the call to search on line 32 defines a path starting with the edge which our segment $X \xrightarrow{+} Y$ ends with; therefore the paths can be combined. By so doing from the bottom of the call stack to the top we construct a unique path from `START`. □

For brevity, we will hereafter refer to the canonical path constructed in the manner of lemma B.1 corresponding to the stack of calls to `Search` when an edge e is first encountered as $CP(e)$. Every edge in the CFG is encountered exactly once by `Search`, so $CP(e)$ exists and is unique for every edge e in the CFG.

Lemma B.2. *SSI form property 2.1 (ϕ-function naming) holds for variables renamed according to Algorithm A.2.*

Proof. We restate SSI form property 2.1 for reference:

> For every node X containing a definition of a variable V in the new program and node Y containing a use of that variable, there exists at least one path $X \xrightarrow{+} Y$ and no such path contains a definition of V other than at X.

We consider the canonical path $CP(\langle Y', Y\rangle) = \texttt{START} \xrightarrow{*} Y' \to Y$ for some use of a variable v at Y, constructed according to lemma B.1 from a stack trace of calls to `Search`. is encountered. This path is unique, although more than one canonical path may terminate at Y at nodes with more than one predecessor. These paths are distinguished by the incoming edge to Y.[11] We identify each operand v_i of a ϕ-function with the appropriate incoming edge e to ensure that $CP(e)$ is well defined and unique in the context of a use of v_i.

The canonical path $\texttt{START} \xrightarrow{+} Y$ must contain X, a definition of v, if Y uses a variable defined in X, as `Search` renames all definitions (in lines 5, 9, and 24) and destroys the name mapping in $\mathcal{E}$ just before it returns. The call to `Search` which creates the definition of v must therefore always be on the stack, and thus in the path $CP(\langle Y', Y\rangle)$, for any use to receive a the name v. Note that this is true for ϕ-functions as well, which receive names when the appropriate incoming edge $\langle Y', Y\rangle$ is traversed, not necessarily when the node Y containing the ϕ-function is first encountered.

We have proved that $\texttt{START} \xrightarrow{+} X \xrightarrow{+} Y$ exists; now we must prove that no other path from X to Y contains a definition of v. Call this other definition D. Obviously D cannot be on our canonical path $\texttt{START} \xrightarrow{+} X \xrightarrow{+} Y$, or line 24 would have caused Y to use a different name. But as we just stated, all variable name mappings done by D will be removed when the call to `Search` which touched D is taken off the call stack. So D must be on the call stack, and thus on the canonical path; a contradiction. Since assuming the existence of some other path $X \xrightarrow{+} Y$ containing a definition of v leads to contradiction no other such path may exist, completing the proof of the lemma. □

[11] Note that the notation $\langle N, N'\rangle$ for denoting edges does not always denote an edge unambigiously; imagine a conditional branch where both the `true` and `false` case lead to the same label. In such cases an additional identifier is necessary to distinguish the edges. Alternatively, one may split such edges to remove the ambiguity. We treat edges as uniquely identifiable and leave the implementation to the reader.

Lemma B.3. *SSI form property 2.2 (σ-function naming) holds for variables renamed according to Algorithm A.2.*

Proof. We restate SSI form property 2.2 for reference:

> For every pair of nodes X and Y containing uses of a variable V defined at node Z in the new program, either every path $Z \xrightarrow{+} X$ must contain Y or every path $Z \xrightarrow{+} Y$ must contain X.

Let us assume there are paths $Z \xrightarrow{+} X$ and $Z \xrightarrow{+} Y$ violating this condition; that is, let us chose nodes X and Y which use V and Z defining V such that there exists a path P_1 from Z to X not containing Y and a path P_2 from Z to Y not containing X. By the argument of the previous lemma, there exists a canonical path $P_3 = CP(e)$ from `START` to X through Z corresponding to a stack trace of `Search`; note that P_3 need not contain P_1. There are two cases:

Case I: P_3 does not contains Y. Then there is some last node N present on both $P_2 : Z \xrightarrow{*} N \xrightarrow{+} Y$ and $P_3 : \texttt{START} \xrightarrow{+} Z \xrightarrow{*} N \xrightarrow{+} X$. By the path-convergence criterion for σ-functions, this node N requires a σ-function for V. If $N \neq Z$ then line 5 of Algorithm A.2 would rename V along P_3 and X would not use the same variable Z defined; if $N = Z$, then line 9 would have ensured that X and Y used different names. Either case contradicts our choices of X, Y, and Z.

Case II: P_3 does contain Y. Then consider the path $\texttt{START} \xrightarrow{+} Z \xrightarrow{+} Y$ along P_3, which does not contain X. The argument of case I applies with X and Y reversed.

Any assumed violation of property 2.2 leads to contradiction, proving the lemma. □

Every path $CP(e)$ corresponds to a execution state in a call to `Search` at the point where e is first encountered. The value of the environment mapping $\mathcal{E}$ at this point in the execution of Algorithm A.2 we will denote as $\mathcal{E}^e$. For a node N having a single predecessor N_p and single successor N_s, we will denote $\mathcal{E}^{\langle N_p, N\rangle}$ as $\mathcal{E}^N_{\text{before}}$ and $\mathcal{E}^{\langle N, N_s\rangle}$ as $\mathcal{E}^N_{\text{after}}$. It is obvious that $\mathcal{E}^{N_p}_{\text{after}} = \mathcal{E}^N_{\text{before}}$ and $\mathcal{E}^N_{\text{after}} = \mathcal{E}^{N_s}_{\text{before}}$ when N_p and N_s, respectively, are also single-predecessor single-successor nodes.

Lemma B.4. *SSI form property 2.3 (correctness) holds for variables renamed according to Algorithm A.2. That is, along any possible control-flow path in a program being executed a use of a variable V_i in the new program will always have the same value as a use of the corresponding variable V in the original program.*

Proof. We will use induction along the path $N_0 \to N_1 \to \ldots \to N_n$. We consider $e_k = \langle N_k, N_{k+1}\rangle$, the $(k+1)$th edge in the path, and assume that, for all $j < k$, each variable V

in the original program agrees with the value of $\mathcal{E}^{e_j}[V] = V_i$ in the new program. We show that $\mathcal{E}^{e_k}[V]$ agrees with V at edge e_k in the path.

Case I: $k = 0$. The base case is trivial: the START node (N_0) contains no statements, and along each edge e leaving start $\mathcal{E}^e[V] = V_0$. By definition V_0 agrees with V at the entry to the procedure.

Case II: $k > 0$ and N_k has exactly one predecessor and one successor. If N_k is single-entry single-exit, then it is not a ϕ- or σ-function. As an ordinary assignment, it will be handled by lines 20 to 24 of Algorithm A.3 on page 13. By the induction hypothesis (which tells us that the uses at N_k correspond to the same values as the uses in the original program) and the semantics of assignment, the mapping $\mathcal{E}^{N_k}_{\text{after}}$ is easily verified to be valid when $\mathcal{E}^{N_k}_{\text{before}}$ is valid. Thus the value of every original variable V corresponds to the value of the new variable $\mathcal{E}^{N_k}_{\text{after}}[V] = \mathcal{E}^{e_k}[V]$ on e_k.

Case III: $k > 0$ and N_k has multiple predecessors and one successor. In this case N_k may have multiple ϕ-functions in the new program, and N_k has no statements in the original program. Thus the value of any variable V in the original program along edge e_k is identical to its value along edge e_{k-1}. We need only show that the value of the variable $\mathcal{E}^{e_{k-1}}[V]$ is the same as the value of the variable $\mathcal{E}^{e_k}[V]$ in the new program. For any variable V not mentioned in a ϕ-function at N_k this is obvious. Each variable defined in a ϕ-function will get the value of the operand corresponding to the incoming control-flow path edge. The relevant lines in Algorithm A.3 start with 13 and 14, where we see that the operand corresponding to edge e_{k-1} of a ϕ-function for V correctly gets $\mathcal{E}^{e_{k-1}}[V]$. At line 5, we see that the destination of the ϕ-function is correctly $\mathcal{E}^{e_k}[V]$. Thus the value of every original variable V correctly correponds to $\mathcal{E}^{e_k}[V]$ by the induction hyptothesis and the semantics of the ϕ-functions.

Case IV: $k > 0$ and N_k has one predecessor and multiple successors. Here N_k may have multiple σ-functions in the new program, and is empty in the original program. The argument goes as for the previous case. It is obvious that variables not mentioned in the σ-functions correspond at e_k if they did at e_{k-1}. For variables mentioned in σ-functions, line 18 shows that operands correctly get $\mathcal{E}^{e_{k-1}}[V]$ and line 9 shows that the destination corresponding to e_k correctly gets $\mathcal{E}^{e_k}[V]$. Therefore the values of original variables V correspond to the value of $\mathcal{E}^{e_k}[V]$ by the induction hypothesis and the semantics of the σ-functions.

Therefore, on every edge of the chosen path, the values of the original variables correspond to the values of the renamed SSI form variables. The value correspondence at the path endpoint (a use of □

C Optimistic and Pessimistic Algorithms

In our experience, optimistic algorithms tend to have poor time bounds because of the possibility of input graphs like the one illustrated in Figure 18. Proving that all but two nodes require ϕ- and/or σ-functions for the variable a in this example seems to inherently require $O(N)$ passes over the graph; each pass can prove that ϕ- or σ-functions are required for only those nodes adjacent to nodes tagged in the previous pass. Starting with the circled node, the ϕ- and σ-functions spread one node left on each pass. On the other hand, a pessimistic algorithm assumes the correct answer at the start, fails to show that any ϕ- or σ-functions can be removed, and terminates in one pass.

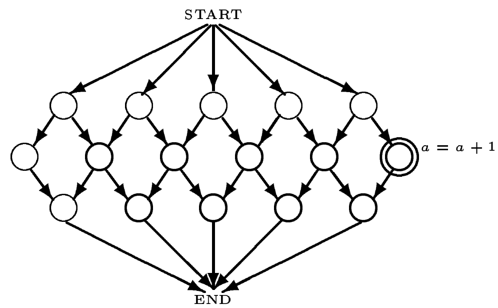

Figure 18: A worst-case CFG for "optimistic" algorithms.