The Static Single Information Form

by

C. Scott Ananian

B.S.E. Electrical Engineering

Princeton University, 1997

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

September 3, 1999

Author____

Department of Electrical Engineering and Computer Science

September 3, 1999

Certified by____

Martin Rinard

Thesis Supervisor

Accepted by____

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

The Static Single Information Form
by
C. Scott Ananian

Submmitted to the
Department of Electrical Engineering and Computer Science

September 3, 1999

In partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

The Static Single Information (SSI) form is a compiler intermediate representation that allows efficient sparse implementations of predicated analysis and backward dataflow algorithms. It possesses several attractive graph-theoretic properties which aid in program analysis. An extension to SSI form, SSI^+, is also presented, along with a complete executable abstract semantics for the representation. Applications to abstract interpretation and hardware compilation are discussed.

The SSI form has been implemented on the FLEX compiler infrastructure, and it has been used to implement several analyses and optimizations. Details on these predicated analysis techniques are presented, as well as data from the practical implementation.

Thesis Supervisor: Martin Rinard
Title: Professor, Laboratory for Computer Science

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This paper introduces a compiler intermediate representation: Static Single Information (SSI) form. This IR is the core of the FLEX compiler project, which is primarily investigating intelligent compilation techniques for distributed systems. This thesis, in presenting the IR, attempts to keep both the mathematician and the programmer in mind. SSI form has both a rigorous mathematical semantics and a factored form which aids efficient implementation of advanced analyses. I believe that it effectively straddles the gap between dataflow-oriented, graph-structured, and control-flow driven IRs, while maintaining the sparsity needed to achieve practical efficiency. The construction algorithms are linear in the size of the program.

Our discussion of the Static Single Information form will be at times tied to the source language of the FLEX compiler, Java. Unlike many abstract IRs, the choices made in the design of SSI form have been dictated by the necessities of compiling a real-world imperative language. Java, however, has several theoretical properties that make program analysis more tractable. In particular, we mention here Java's strict constraints on pointer variables. Pointers in earlier languages such as C can be abused in many ways that Java disallows.

Ultimately, the choice of compiler internal representation is fundamental. Advances in IRs translate into advances in compilers. SSI form represents a clean and simple unification of many extant ideas, and our hope is that it will allow the FLEX compiler to achieve a similar integration of practical implementation and mathematical elegance.

2 Context and goals

Strong et al. [40][1] first advocated the use of compiler intermediate representations in a 1958 committee report. Their idealistic "universal intermediate language" was called UNCOL. Thirty years later, the Static Single Assignment (SSA) form was introduced by Alpern, Rosen, Wegman and Zadeck as a tool for efficient optimization in a pair of POPL papers [2, 35], and three years after that Cytron and Ferrante joined Rosen, Wegman, and Zadeck in explaining how to compute SSA form efficiently in what has since become the "canonical" SSA paper [10]. Johnson and Pingali [20] trace the development of SSA form back to Shapiro and Saint in [37], while Havlak [17] views ϕ-functions as descendants of the "birthpoints" introduced in [34].

Despite industry adoption of SSA form in production compilers [8, 9], academic research into alternative representations continues. Recent proposals have included Value Dependence Graphs [45], Program Dependence Webs [5], the Program Structure Tree [19], DJ graphs [39], and Depedence Flow Graphs [20].

In comparison to these representations, the dominant characteristics of our Static Single Information form may be summarized as follows:

- It names information units.
- It is complete.
- It is simple.
- It is efficient.
- It has no explicit control dependencies.

[1] Attribution by Aho [1].

- It supports both forward and reverse dataflow analyses.

SSI form is used as an IR for the FLEX compiler for the Java programming language, which informs some of these design decisions. The FLEX compiler does deep analysis and will support hardware/software co-design. SSI addresses these needs, concentrating on analysis rather than optimization. We will address each design point in turn.

It names information units. SSA form (which we will describe further in section 4) assigns unique names to unique *static values* of a variable. However, it ignores the value information which may be added to a variable at program branch points. SSI form renames variable at branch points, which allows us to associate unique names with unique *information* about static values. For example, a program may test the value of an integer against zero before using it as a divisor. After the branch on the tested predicate, it is possible to make statements about values (regarding equality or inequality to zero) which were impossible to make previously. SSI form allows us to exploit this additional information.

It is complete. By this we mean that there exists an executable semantics for the IR that does not require the use of information external to the IR. The original SSA form—and most derivatives—require use of the original program control flow graph during analysis, translation, or direct execution. In fact, ϕ-functions are intimately tied with the precise input edge structure of the control flow graph, and switch nodes (where control flow splits) are undecipherable without referring to the control flow graph.

In practice, this seems not a great disadvantage—it merely forces us to maintain a mapping of SSA statements to nodes (equivalently, basic blocks) of the original control flow graph. But maintaining this correspondence complicates editing the IR. Also, it complicates the interpretation of the program as a set of simultaneous equations, which SSI form will allow us to do. Finally, explicit control flow may limit the available parallelism of the program.

SSI^+, as it will be presented in section 7, overcomes these difficulties and presents a *complete* representation of program meaning as a set of simultaneous equations, without resort to graph information.

It is simple. A bestiary of new ϕ-like functions have been introduced in the past decade, including μ-, γ-, and η-functions in [5, 43], ψ- and π-functions in [24], interprocedural ϕ-functions in [26], μ- and χ-functions in [9], μ- and η-functions in [14],[2] and Λ-functions in [27], among others.Some of these are orthogonal to our work—the techniques of [24] can be used to extend SSI form to explicitly parallel source languages, and those of [9] to languages with local variable aliasing (absent in Java). Our goal is to achieve minimal conceptual complexity in SSI form; that is, to introduce the minimum set of ϕ-like functions necessary to represent the "interesting" properties of the compiled program.

It is efficient. Construction of SSI form should be fast, and space requirements should be reasonable. The original SSA algorithms required $O(E + V_{SSA}|DF| + NV_{SSA})$ time.[3] This bound was dominated by the time and space required to construct the dominance frontier, as $|DF|$, the size of the dominance frontier, could be $O(N^2)$ for common cases. Taking the dominant term, we abbreviate the time complexity of the Cytron's SSA-construction algorithm as $O(N^2V)$.

Our algorithms do not require the construction of a dominance frontier—building on recent work on efficient SSA construction in this regard—and run in so-called "linear" time. A more detailed analysis will be given in section 5.4, but suffice for now to say that our construction and analysis

[2] Compare to [5, 43].

[3] See section 3 for definitions of the variables used in the complexity bounds of these two paragraphs.

algorithms are efficient.[4]

All explicit control dependencies are eliminated. Some researchers (including [4] and [32]) view control dependence as a fundamental property of the CFG, and [5, 4] suggest that accurate knowledge of control-dependence relations is the sole key to automatic parallelization. Often, incomplete intermediate representations[5] are augmented with control-dependence edges to express proper program semantics—see [20] on DFGs and [45] on VDGs, for example.

Unfortunately, explicit control-flow edges tend to serialize computation more than strictly necessary. Figure 7.1 on page 75, for example, contains two parallel loops which would be serialized by the explicit control dependency between them. Prior work often focused on fine-grain intra-loop parallelism and ignored this coarser inter-loop parallelism.[6] Our objective in this work is to fully utilize coarse parallelism by removing source-language control-dependency artifacts.

It is efficient for both forward and backward dataflow analyses. It is often observed that traditional SSA form cannot handle backward dataflow analysis. Johnson and Pingali note this, and suggest *anticipatability* as an example of a backwards dataflow analysis where their dependence flow graph representation betters SSA form [20]. Lo et al. suggest the use of an "SSU" form to address much the same issue [27]. There are in fact many analyses where both use and definition information is utilized, and where dataflow in both forward and reverse directions occurs. SSI form is able to handle both of these cases, as we demonstrate in section 6.1.

---

[4] Dhamdhere [12] quite correctly states that Cytron's original algorithm has a worst-case time bound of $O(N^3)$. This is also true for our algorithms. However, these worst-case time bounds are not tight; we will present experimental evidence that run times on real programs are $O(N)$.

[5] See page 9 for our definition of "completeness" in an IR.

[6] We discuss the dataflow-architecture work of Traub [42] in particular in section 7.5.

3 Definitions

We next provide some definitions. Our complexity metrics will usually be in terms of the following variables:

N is the number of nodes in the program control flow graph. Each node represents either a single statement or a basic block; the difference is unimportant for complexity metrics.

E is the number of edges in the program control flow graph. For most programs E is reasonably assumed to be $O(N)$, since most nodes have either one or two successors (simple assignments and conditional branches, respectively). Unusual use of computed-goto and `switch` statements may invalidate this assumption; but in these cases E is generally a better metric of program "complexity" than N. For this reason, we will case $O(E)$ "linear in program size".

V is the number of variables in the program.

U is the total number of variable uses in the program.

As the transformations we will describe split and rename variables, we will use subscripts to denote the number of variables, uses, or definitions in a particular transformed version of a program. For example, U_{SSA} is the number of uses in the SSA form (see section 4) of a program. When it is necessary to explicitly denote a metric on the untransformed program, a zero subscript will be used; for example, V_0.

Graphs will be directed unless specified otherwise. If X and Y are nodes in some graph G, an edge from X to Y is written $X \to Y$. A path $X = s_0 \to s_1 \to \ldots \to s_n = Y$ is written $X \xrightarrow{+} Y$. A ***simple*** path is one in which all the nodes s_i in it are distinct.

Control-flow graphs are assumed to be connected, and to contain unique START and END nodes marking procedure entry and exit points, respectively. To ensure that graphs representing infinite loops are connected, an edge will typically exist between the START and END nodes. The presence of unique START and END nodes ensures that both the dominance and post-dominance relation define trees rooted at START and END, respectively.

For simplicity, we will assume that every node in the control-flow graph with one successor and one predecessor contains exactly one statement. A node with no predecessors and a node with no successors (START and END) are empty; they contain no statements. Nodes with multiple successors or multiple predecessors are also empty for conventional program representations, but may contain multiple ϕ- or σ-function assignment statements in the SSA and SSI forms we will discuss. No node may contain both multiple predecessors and multiple successors.

The symbol $\sqcap$ will be used for the dataflow "meet" operator. The operator $\sqsubseteq$ is the partial ordering relation for a lattice, and $x \sqsubset y$ iff $x \sqsubseteq y$ and $x \neq y$.

4 Static Single Assignment form

Static Single Information (SSI) form derives many features from Static Single Assignment (SSA) form, as described by Cytron in [10]. To provide context for our definition of SSI form in section 5, we review SSA form.

4.1 Definition of SSA form

Static Single-Assignment form is a sparse program representation in which each variable has exactly one definition point. As a consequence, only one assignment can reach each use, which means that SSA form can be viewed

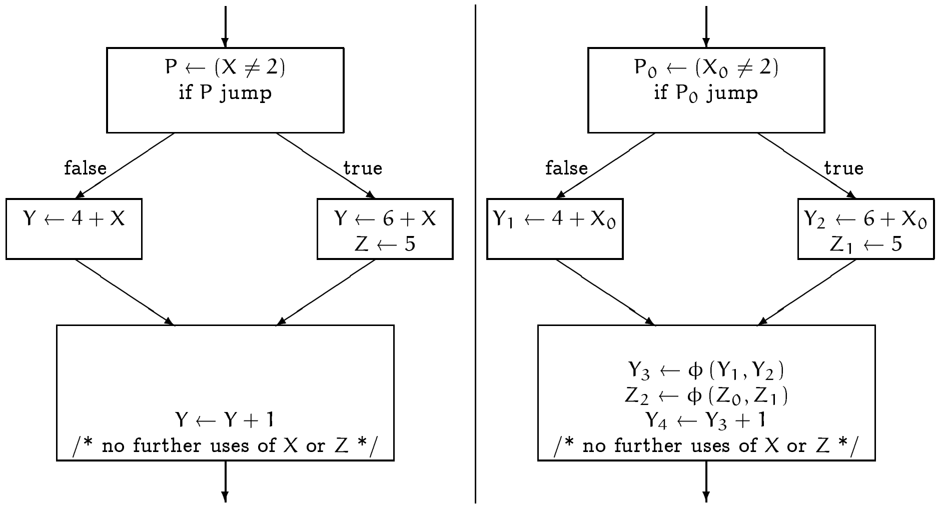


Figure 4.1: A simple program (left) and its single assignment version (right).

as a type of sparse ***def-use chain*** [1].

For straight-line code, the SSA transformation is straightforward: each assignment to a variable is given a unique name (conventionally indicated by the use of a subscripted version of the original variable name) and each use is renamed to match its reaching definition. Special ϕ-***functions*** must be inserted at join points to preserve the single-assignment property. These ϕ-functions have the form $\nu_0 \leftarrow \phi(\nu_1, \nu_2)$ and perform an assignment according to the path by which control flow reaches the ϕ-function. Figure 4.1 shows a simple program and its SSA form; the ϕ-function $Y_3 \leftarrow \phi(Y_1, Y_2)$ in the SSA version on the right assigns Y_3 the value of Y_1 if control flow reaches it along the false branch of the `if` statement. If the true branch is taken, Y_3 will get the value of Y_2 at the ϕ-function.

Formally, a program is said to be in SSA form if the following three conditions hold:

1. If two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at a node Z, and nodes X and Y contain assignments to [a variable] V (in the original program), then a trivial ϕ-function $V \leftarrow \phi(V, \ldots, V)$ has been inserted at Z (in the new program).

2. Each mention of V in the original program or in an inserted ϕ-function has been replaced by a mention of a new variable V_i, leaving the new program in SSA form.

3. Along any control flow path, consider any use of a variable V (in the original program) and the corresponding use of V_i (in the new program). Then V and V_i have the same value.

This formulation of this definition is due to Cytron et al. [11]. Note that the definition does not prohibit "extra" ϕ-functions not strictly required by condition 1.

4.2 Minimal and pruned SSA forms

Cytron et al. [11] defines ***minimal SSA form*** as an SSA form using the smallest number of ϕ-functions such that the above three conditions hold. The SSA form in the previous example (Figure 4.1 on the facing page) is minimal.

A variation on minimal SSA form, called ***pruned form***, avoids placing ϕ-functions which define variables which are never used. The ϕ-functions in pruned form are a subset of those in minimal form, and as such note that pruned form does not strictly satisfy the given SSA criteria. In most cases, the more regular properties of minimal SSA form outweigh the pruned form's slight increase in space efficiency. Choi, Cytron, and Ferrante [7] give a formal definition and construction algorithm for pruned SSA.

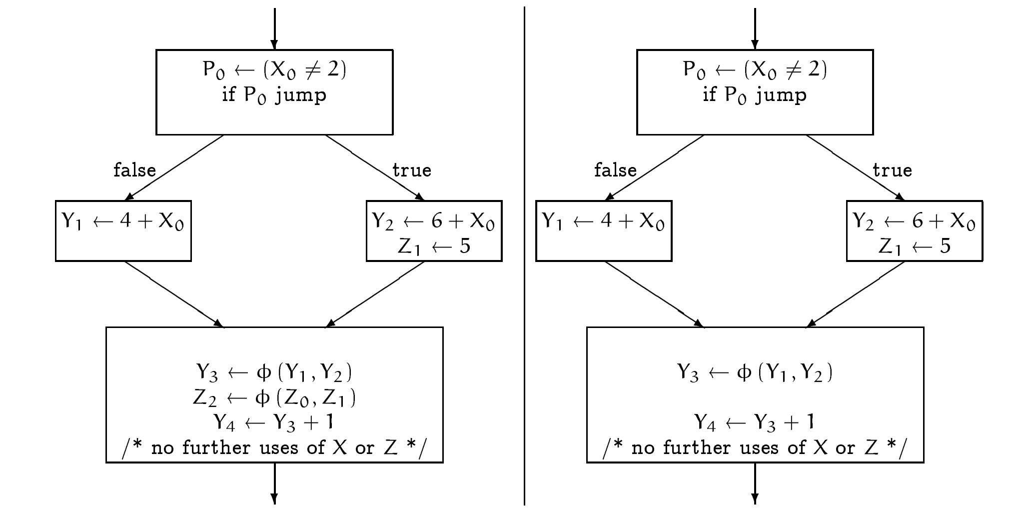


Figure 4.2: Minimal (left) and pruned (right) SSA forms.

Figure 4.2 compares minimal and pruned SSA form for our example program.

5 Static Single Information form

SSI form extends SSA form to achieve symmetry for both forward and reverse dataflow. SSI form recognizes that information about variables is generated at branches and generates new names at these points. This provides us with a one-to-one mapping between variable names and information about the variables at each point in the program. Analyses can then associate information with variable names and propagate this information efficiently and directly both with and against the control-flow direction.

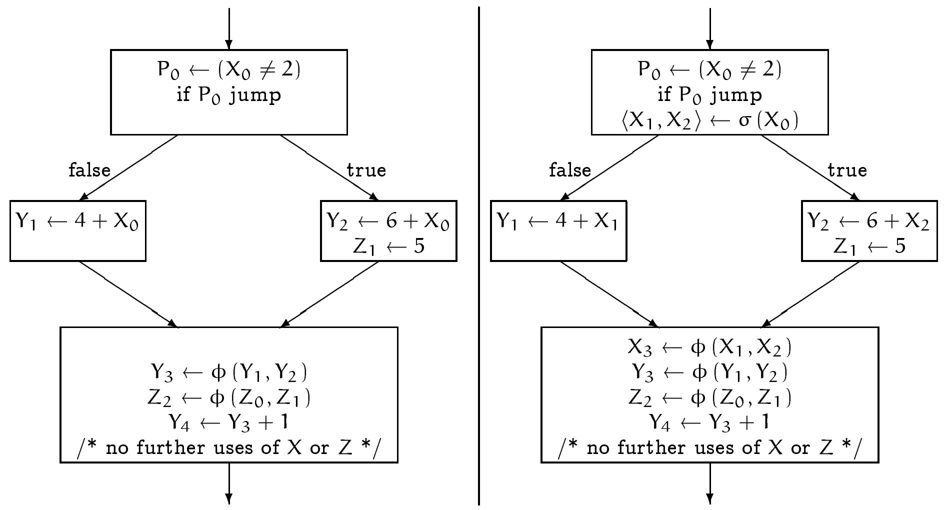


Figure 5.1: A comparison of SSA (left) and SSI (right) forms.

5.1 Definition of SSI form

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.

Figure 5.1 compares the SSA and SSI forms for the example of Figure 4.1. Note that X is renamed at the conditional branch, allowing the compiler to distinguish between X_1 (which is always the constant 2) from X_2 (which is never equal to 2).

Formally, a program transformation to SSI form satisfies the following conditions:

1. If two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ exist having only the node Z where they converge in common, and nodes X and Y contain either assignments to a variable V in the original program or a ϕ- or σ-function for V in the new program, then a ϕ-function for V has been inserted at Z in the new program. [Placement of ϕ-functions.]

2. If two nonnull paths $Z \xrightarrow{+} X$ and $Z \xrightarrow{+} Y$ exist having only the node Z where they diverge in common, and nodes X and Y contain either uses of a variable V in the original program or a ϕ- or σ-function for V in the new program, then a σ-function for V has been inserted at Z in the new program. [Placement of σ-functions.]

3. 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. [Naming after ϕ-functions.]

4. 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. [Naming after σ-functions.]

5. For the purposes of this definition, the `START` node is assumed to contain a definition and the `END` node a use for every variable in the original program. [Boundary conditions.]

6. Along any possible control-flow path in a program being executed consider any use of a variable V in the original program and the corresponding use of V_i in the new program. Then, at every occurance of the use on the path, V and V_i have the same value. The path need not be cycle-free. [Correctness.]

As with the SSA conditions, this definition does not prohibit "extra" ϕ- or σ-functions not required by conditions 1 and 2.

Property 5.1. *There exists exactly one reaching definition of* V *at every non-*ϕ*-function use of* V *in the new program.*

Proof. Offner [29] defines a reaching definition as follows:

> A definition of a variable v ***reaches*** the point P in the program iff there is a path from the definition to P on which...there is no other definition of v....

From this definition and condition 3 we directly obtain the property. □

Note that condition 3 and this property do not require there to be exactly one definition of any variable V, just that at every use only a single definition is relevant. The renaming algorithm we will present enforces the stricter single-definition constraint.

Property 5.2. *Every cycle-free path* $S \xrightarrow{+} Y$ *from the* START *node to a node* Y *containing a non-*ϕ*-function use of a variable must contain exactly one node* X *defining that variable in the new program. Likewise, every path* $X \xrightarrow{+} E$ *from a node* X *containing a non-*σ*-function definition of a variable to the* END *node must contain every node* Y *which is a use of that variable in the new program.*

Proof. Let us call the variable v. Conditions 5 and 6 ensure that there exists at least one definition node X for v from which Y is reachable—conditions 5 and 6 substitute the START node, from which every node is reachable, for any use of v not reachable by some other definition in the original program. So assume this definition node X exists, but is not on the path $S \xrightarrow{+} Y$. Then $X \xrightarrow{+} Y$ and $S \xrightarrow{+} Y$ must have some earliest node

N in common. But N must then have a ϕ-function for v by condition 1, which violates either our choice of Y as a non-ϕ-function use (if $N = Y$) or else condition 3 which prohibits definitions other than at X. If $S \xrightarrow{+} Y$ contains more than one node X_i defining v, then the path $X_0 \xrightarrow{+} Y$ between the first and Y also violates condition 3. So $S \xrightarrow{+} Y$ must contain exactly one definition X of v.

The second part is symmetric. Assume there exists some node Y using v which is not contained on some path $X \xrightarrow{+} E$. The path $X \xrightarrow{+} Y$ must exist by conditions 3 and 5. And $X \xrightarrow{+} E$ and $X \xrightarrow{+} Y$ must have some final node N in common, which must have a σ-function for v by condition 2. The case $N = X$ violates the choice of X as a non-σ-function definition. But if $N \neq X$, then condition 3, which prohibits paths with multiple definitions, is violated. Thus $X \xrightarrow{+} E$ must contain every use of v. □

Property 5.3. *Every definition of a variable V dominates all non-ϕ-function uses of V and every use of V post-dominates any non-σ-function reaching definition of V in the new program.*

Proof. The dominance relation is defined in Offner [29] as:

> If x and y are two elements in a flow graph G, then x ***dominates*** y (x is a ***dominator*** of y) iff every path from s [START] to y includes x.

Post-dominance is the dual on a flow graph with edges reversed: x post-dominates y iff every path from END to y includes x.

The previous property showed that every path from START to a non-ϕ-function use contained a unique definition node X. If two paths from START to Y contained different definition nodes X_i, then Y would be a ϕ-function, which it was chosen not to be. So every non-ϕ-function use is dominated by the single definition node. Likewise the previous property showed that every path from a non-σ-function definition to END must include every use; therefore every use post-dominates a non-σ-function definition. ☐

5.2 Minimal and pruned SSI forms

Minimal and *pruned* SSI forms can be defined which parallel their SSA counterparts. *Minimal* SSI form would have 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.[7] Figure 5.2 on the next page compares minimal and pruned SSI form for our 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 conditions 1 and 2 of the definition. In practice, a subtractive definition of pruned form — generate minimal form and then removed 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 conditions 1 and 2 must also satisfy: "and there exists a path from $Z \xrightarrow{+} U$ to a U, a use of V in the original program, which does not contain another definition of V."

2. The boundary condition 5 at 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

[7] 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 control-flow splits which generate additional value information.

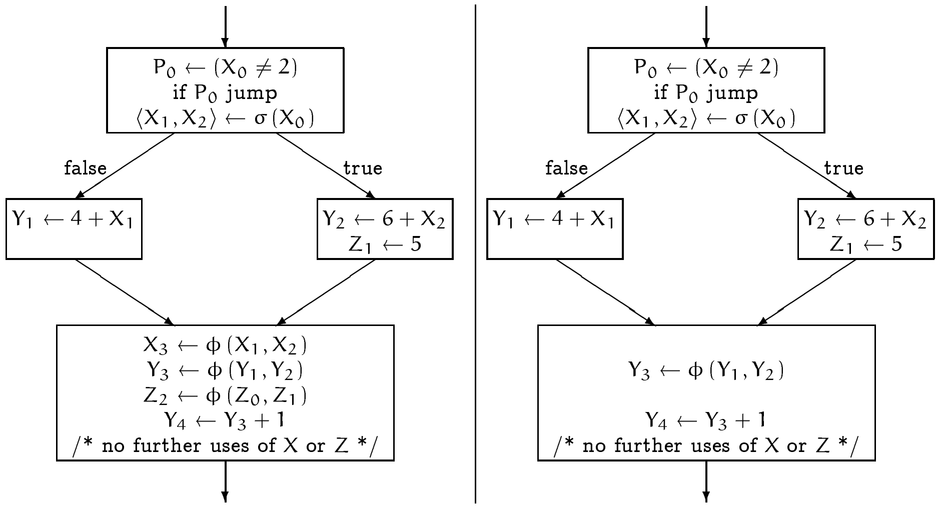


Figure 5.2: Minimal (left) and pruned (right) SSI forms.

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 path $Z \xrightarrow{+} U$ as mandated by amendment 1, where amendment 2 lets $U = \text{END}$ for variables live exiting the procedure and thus usefully defined.

Property 5.4. *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 conditions 1 and 2 can be found and neither a ϕ- or σ-function can be placed. Amendment 2 ensures this holds true at boundaries. □

5.3 Fast construction of SSI form

The most common construction algorithm for SSA form [11] 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 [38] and the dependence flow graph [20] 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 [19]. We will review the algorithms involved, as their published descriptions [18] contain a number of errors.

We begin with a few definitions from [19].

Definition 5.1. *Edges* a *and* b *are said to be* **edge cycle-equivalent** *in a graph iff every cycle containing* a *contains* b*, and vice-versa. Similarly, two nodes are said to be* **node cycle-equivalent** *iff every cycle containing one of the nodes also contains the other.*

Definition 5.2. *A* **SESE region** *in a graph* G *is an ordered edge pair* $\langle a, b\rangle$ *of distinct control flow edges* a *and* b *where*

1. a *dominates* b*,*
2. b *postdominates* a*, and*
3. *every cycle containing* a *also contains* b *and vice-versa.*

Edges a *and* b *are called the* **entry** *and* **exit** *edges, respectively.*

Definition 5.3. *A SESE region* $\langle a, b\rangle$ *is* **canonical** *provided*

1. b *dominates* b' *for any SESE region* $\langle a, b'\rangle$*, and*
2. a *postdominates* a' *for any SESE region* $\langle a', b\rangle$*.*

We will give time bounds in terms of N and E, the number of nodes and edges of the control-flow graph, respectively. Placement of ϕ- and σ-functions is also dependent on V, the number of variables in the program. Since SSI renaming increases the number of variables, we will use V_0 and V_{SSI} to indicate the number of variables in the original program and SSI form, respectively.

Note that V is $O(N)$ at most, since our representation only allows a constant number of variable definitions per node. Typically V_0 will be much smaller than N, but V_{SSI} need not be. Also E may be as large as $O(N^2)$, but in most control-flow graphs is $O(N)$ instead, as node arities are typically limited by a constant.

5.3.1 Cycle-equivalency

The identification of SESE regions begins by computing the cycle-equivalency of the edges in the program control flow graph. The cycle-equivalency algorithm works on undirected graphs, so we prepare the directed control flow graph G as follows:

1. **Add an edge from** `END` **to** `START` **in G.** It is common practice to add an edge from `START` to `END` in order to root the control dependence graph at `START` [10]. However, our goal is not rooted control dependence but to make the control flow graph into a single strongly connected component; for this reason the direction of the edge is from `END` to `START` instead.

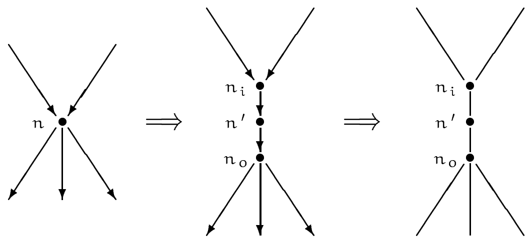


Figure 5.3: Transformation from directed to undirected graph (from [18]).

2. **Create an equivalent undirected graph.** Johnson et al. prove that the node expansion illustrated in Figure 5.3 results in an undirected graph with the same cycle-equivalency properties as the original directed graph. More precisely, nodes a and b in directed graph G are cycle-equivalent if and only if nodes a' and b' are cycle-equivalent in transformed undirected graph G'. The nodes n_i and n_o generated by the expansion are termed *not representative*; the node n' in G' is said to be *representative* of node n in G. Obviously, this correspondence must be recorded during the transformation so we may properly attribute the cycle-equivalency properties of n' to n later.

3. **Perform a pre-order numbering of nodes in G'.** This is done with a simple depth-first search of G'. When we visit a node a_i or a_o, we prefer to visit a' before any other neighbor. This ensures that representative nodes are interior nodes in the DFS spanning tree. The `START` node is numbered 0, and succeeding nodes in the traversal get increasing numbers. Thus low-numbered nodes are closest to `START` and we will call them "highest" in the DFS spanning tree.

The above steps form an undirected graph G' from the control-flow graph G. The remainder of the cycle-equivalency algorithm is presented

Data type `BracketList`:

create(): BracketList : Make an empty BracketList structure
size(bl:BracketList): integer : Number of elements in BracketList structure
push(bl:BracketList, e:bracket): BracketList : Push **e** on top of **bl**
top(bl:BracketList): bracket : Topmost bracket in **bl**
delete(bl:BracketList, e:bracket): BracketList : Delete **e** from **bl**
concat(bl1,bl2:BracketList): BracketList : Concatenate **bl1** and **bl2**

Operations on nodes:

Number(n:node): integer : DFS preorder number of node
NQClass(n:node): integer : Cycle-equivalency class of node
BList(n:node): BracketList : List of brackets of node
Hi(n:node): integer : Highest destination node of any edge originating from a descendant of node **n**

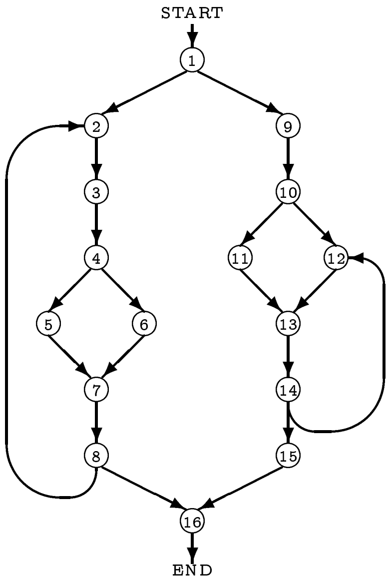
Operations on edges:

EQClass(e:node): integer : Cycle-equivalency class of edge
RecentSize(e:edge): integer : Size of bracket set when **e** was most recently the topmost bracket for a representative node
RecentClass(e:edge): integer : Cycle-equivalency class number of representative node for which **e** was most recently the topmost bracket.

Figure 5.4: Datatypes and operations for the cycle-equivalency algorithm.

```
Procedure cycle_equiv (G: CFG)
{
  /* Preprocessing */
  G' := Preprocess (G); /* described in text */

  /* Compute CD equivalence classes */
  for each node n of G', in reverse depth-first order, do {
    /* Compute Hi(n) */
    /*  hi0 is highest using backedges only */
    hi0 := min{ Number(t) | (t,n) is a backedge };
    /*  hi1 is highest through children */
    hi1 := min{ Hi(c) | c is a child of n };
|   /*  hi2 is lowest through children */
|   hi2 := max{ Hi(c) | c is a child of n };

    Hi(n) := min{ hi0, hi1 };

    /* Compute BList(n) */
    BList(n) := create ();

    for each child c of n, do
      BList(n) := concat (BList(n), BList(c));

    for each backedge <d, n> from a descendant d of n to n, do
      BList(n) := delete (BList(n), <d, n>);
|   for each capping backedge <d, n> of n, do
|     BList(n) := delete (BList(n), <d, n>);

    for each backedge <n, a> from n to an ancestor a of n, do {
      BList(n) := push (BList(n), <n, a>)
      RecentSize(<n, a>) := -1; /* not a representative node */
    }

    if n has more than one child, then {
      BList(n) := push (BList(n), <n, hi2>); /* capping backedge */
|     RecentSize(<n, hi2>) := -1;
|     add <n, hi2> to capping backedges list of hi2;
    }

    /* Compute NQClass (n) */
    if n is a representative node, then {
      if RecentSize (top (BList(n))) != size (BList(n)), then {
        /* start a new equivalence class */
        RecentSize (top (BList(n))) := size (BList(n));
        RecentClass (top (BList(n))) := new-class-name();
      }
      NQClass (n) := RecentClass (top (Blist(n)));
    }
  } /* for each node */
}
```


Algorithm 5.1: The cycle-equivalency algorithm (corrected from [18]).



$$\langle \mathtt{START}, 1\rangle \equiv_{cq} \langle 16, \mathtt{END}\rangle$$
$$\langle 1, 2\rangle \equiv_{cq} \langle 8, 16\rangle$$
$$\langle 2, 3\rangle \equiv_{cq} \langle 3, 4\rangle \equiv_{cq} \langle 7, 8\rangle$$
$$\langle 4, 5\rangle \equiv_{cq} \langle 5, 7\rangle$$
$$\langle 4, 6\rangle \equiv_{cq} \langle 6, 7\rangle$$
$$\langle 1, 9\rangle \equiv_{cq} \langle 9, 10\rangle \equiv_{cq} \langle 14, 15\rangle \equiv_{cq} \langle 15, 16\rangle$$
$$\langle 10, 11\rangle \equiv_{cq} \langle 11, 13\rangle$$

Figure 5.5: Control flow graph and cycle-equivalent edges.

as Algorithm 5.1 on the preceding page, with the above procedure corresponding to the statement `G':=Preprocess(G)`. The algorithm has been corrected from the published version in [18]; in addition it has been extended to compute both node and edge equivalencies (in effect, merging the algorithm of [19]). Lines modified from the presentation in [18] are indicated in the figure with a vertical bar in the left margin. The datatype `BracketList` and the node and edge properties used in the algorithm are described in Figure 5.4 on page 26. The interested reader is encouraged to consult [18] for additional detail on these data structures and representations. Figure 5.5 shows cycle-equivalent regions in a simple control-flow graph. We use the notation $\langle a, b\rangle \equiv_{cq} \langle c, d\rangle$ to indicate that the CFG edge from node a to node b is edge cycle-equivalent to the edge from node c to node d.

Calculating cycle-equivalent regions is based on a single reverse depth-first traversal of G, so as long as all datatype operations in Figure 5.4 can be completed in constant time (and [18] shows how to do so), this computation is $O(E)$.

5.3.2 SESE regions and the program structure tree

Johnson, Pearson, and Pingali show how to construct a tree structure of nested SESE regions from the cycle-equivalency information in [19]. The cycle-equivalent regions are sorted by dominance using a simple depth-first traversal of the graph, and then canonical SESE regions are found by taking adjacent pairs of edges from the cycle-equivalence classes. Another depth-first search of the CFG suffices to obtain to nesting of these regions, which is represented in a data structure called the *program structure tree*. The algorithm and data structures required are presented in Figure 5.6 and Algorithm 5.2. Figure 5.7 on page 32 shows the SESE regions on the left and program structure tree on the right for the example of Figure 5.5 on the preceding page.[8]

The time complexity for constructing the PST is easily seen to be $O(E)$. Algorithm 5.2 on page 31 begins with a depth first traversal of G to construct an ordered edge list for each cycle-equivalent region; the traversal is $O(E)$ and the list-append operation can be done in constant time. We then iterate through the cycle-equivalence classes and the edge lists of each constructing SESE regions. No edge can be on more than one list, so this step is $O(E)$. Finally, we do a final $O(E)$ depth-first traversal of G, performing the constant-time operations append and `LinkRegion`. All steps are $O(E)$ and their sequential composition is also $O(E)$.

[8] In addition, the regions c, d, e and f, g are *sequentially composed* [19]. However, our SSI construction algorithm doesn't use this property.

Data type `EdgeList`:

size(el:EdgeList): integer : Number of elements in EdgeList structure
head(el:EdgeList): edge : First edge in **el**
tail(el:EdgeList): EdgeList : EdgeList like **el** but missing first element
append(el:BracketList, e:edge): EdgeList : Add **e** to the end of **el**

Data type `Region`:

NewRegion(e1:edge, e2:edge): Region : Creates a new region with entry **e1** and exit **e2** and no parent
Entry(r:Region): Edge : The entry edge of **r**
Exit(r:Region): Edge : The exit edge of **r**
Parent(r:Region): Region : The parent of **r**, or `nil` if none
Nodes(r:Region): NodeList : A list of nodes in **r**
LinkRegion(r1,r2:Region): void : Sets the parent of **r2** to be **r1**

Operations on nodes:

Mark(n:node): boolean : Visited status during DFS
SESE(n:node): Region : The canonical SESE of **n**

Operations on edges:

EntryRegion(e:edge): Region : the region with entry **e**, or `nil` if none exists
ExitRegion(e:edge): Region : the region with exit **e**, or `nil` if none exists

Figure 5.6: Datatypes and operations used in construction of the PST.

```
NestedSESE(G: CFG) =
/* initialize */
for all nodes n of G do
   Mark(n) ← false
for all edges e of G do
   EntryRegion(e) ← nil
   ExitRegion(e) ← nil
/* order edges within cycle-equivalency classes by dominance */
for each edge e of G in depth first order do
   CQList (EQClass(e)) ← append (CQList(EQClass(e)), e)
/* get all canonical SESE regions */
for all equivalency classes q do
   l ← CQList(q)
   while size(l) > 1 do
      r ← NewRegion (head(l), head(tail(l)))
      EntryRegion(Entry(r)) ← r
      ExitRegion(Exit(r)) ← r
/* determine proper nesting of SESE regions */
VisitNode(START, top-region)

VisitNode(n: node, r: Region) =
if Mark(n) = false then
   Mark(n) ← true
   /* record mapping from n to r */
   SESE(n) ← r
   Nodes(r) ← append(Nodes(r), n)
   for each edge ⟨n, n'⟩ from n to n' do
      r1 ← EntryRegion(⟨n, n'⟩)
      r2 ← ExitRegion(⟨n, n'⟩)
      if r = r1 or r = r2 then
         rN ← Parent(r) /* exiting current region */
      else
         rN ← r
      if r1 ≠ nil and r1 ≠ r then
         LinkRegion(rN, r1) /* entering new region */
         rN ← r1
      if r2 ≠ nil and r2 ≠ r then
         LinkRegion(rN, r2) /* entering new region */
         rN ← r2
      VisitNode(n', rN)
```


Algorithm 5.2: Computing nested SESE regions and the PST.

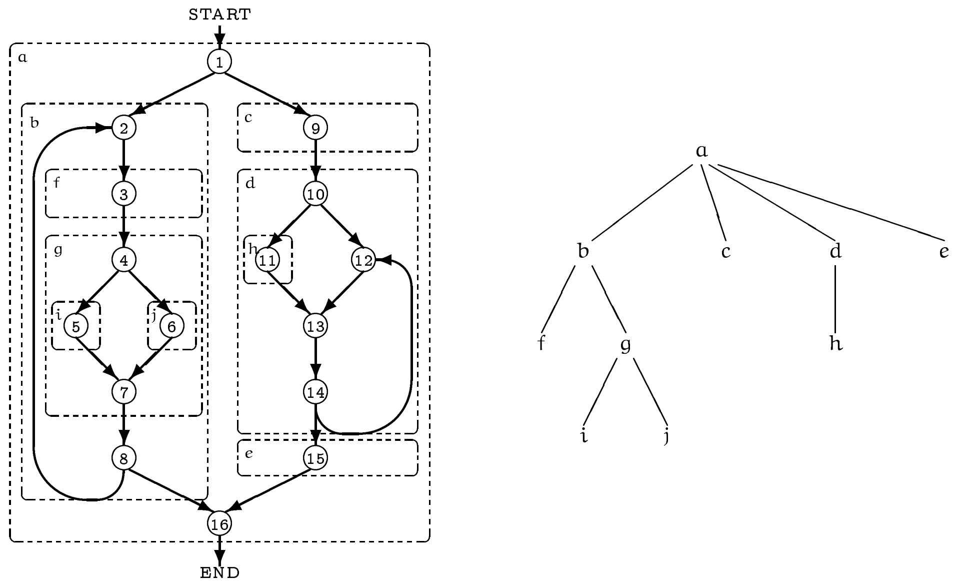


Figure 5.7: SESE regions and PST for the CFG of Figure 5.5 (from [19]).

5.3.3 Placing ϕ- and σ-functions

As with the presentation of SSA form in [11], we split 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 5.3 on the next page. No new node properties or datatypes are required; however, it is parameterized on a function called `MaybeLive`. For minimal SSI form, `MaybeLive` should always return `true`. Faster practical run-time 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 5.3.6 describes a post-processing algorithm to efficiently remove the

```
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 5.3: Placing ϕ- and σ-functions.

excess ϕ- and σ-functions.[9] The remainder of this section will be devoted to a correctness proof of Algorithm 5.3.

Lemma 5.1. *No ϕ-functions (σ-functions) for a variable v are needed in an SESE region not containing a definition (use) of v.*

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 condition 1 of the SSI form definition, 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.

A symmetric argument holds for σ-functions for v, using condition 2 of the SSI form definition, and the fact that there exists one exit edge from the SESE. □

[9] Note that equivalent results could be obtained by adding a ϕ-function for every variable at every merge and a σ-function for every variable at every split, and post-processing. In fact the same time bounds ($O(NV_0)$) would be obtained. There is a large practical difference in actual runtime and space costs, however, which motivates our more efficient approach.

The above lemma justifies line 14 of the algorithm on page 33, which skips over any SESE region not containing a definition (use) of v when placing ϕ-functions (σ-functions) for v.

Lemma 5.2. *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. We will first prove that a node N failing any one of the conditions does not need a ϕ- or σ-function.

- Conditions 1 and 2 of the SSI form definition require node N to be the first convergence (divergence) of some paths $X \xrightarrow{+} N$ and $Y \xrightarrow{+} N$ ($N \xrightarrow{+} X$ and $N \xrightarrow{+} Y$). If the input arity is less than 2 or there is no path from a definition of v, than it fails the ϕ-placement criterion 1. If the output arity is less than 2 or there is no path to a use of v, then it fails the σ-placement criterion 2.

- 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 5.1.

- 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 $D_1 \xrightarrow{+} N$ and $D_2 \xrightarrow{+} N$ containing no nodes in common but

N. We use these paths to construct simple paths $\mathtt{START} \xrightarrow{+} D_1 \xrightarrow{+} N$ and $\mathtt{START} \xrightarrow{+} D_2 \xrightarrow{+} N$. By the definition of a dominator, every path from START to N must contain every D_i. But $D_1 \xrightarrow{+} N$ cannot contain D_2, and if $\mathtt{START} \xrightarrow{+} D_1$ contains D_2, we can make a path $\mathtt{START} \xrightarrow{+} D_2 \xrightarrow{+} N$ which does not contain D_1 by using the D_1-free path $D_2 \xrightarrow{+} N$. 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 conditions 1 and 2 of the SSI form definition and lemma 5.1 that they are sufficient. □

In practice, the conditions of lemma 5.2 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 (for example, a ϕ-function for v at the circled node in Figure 5.8), 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 5.1. *Algorithm 5.3 places all the ϕ- and σ-functions required by conditions 1 and 2 of the SSI form definition.*

Proof. Lemma 5.1 states that the child region exclusion of Algorithm 5.3 does not cause required ϕ- or σ-functions to be omitted. Property 5.4 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

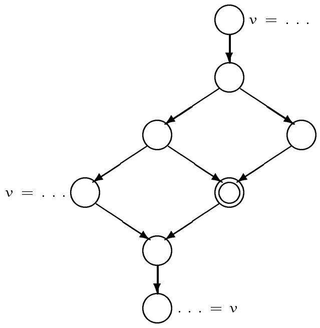


Figure 5.8: An flowgraph where Algorithm 5.3 places ϕ-functions conservatively.

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. □

5.3.4 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 $O(E + N^2)$ [7], which is likely to be more expensive than the rest of the SSI form conversion. Luckily, Kam and Ullman [21], in conjunction with an empirical study by Knuth [23], 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.

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

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

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

Algorithm 5.4: SSI renaming algorithm.

5.3.5 Variable renaming

Algorithm 5.4 performs variable renaming on a flow-graph with placed ϕ- and σ-functions in a single depth-first traversal. When the algorithm is complete, the control flow-graph will be in proper SSI form. The variable renaming algorithm requires an `Environment` datatype which is defined in Figure 5.9. 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 5.4 is a simple depth-first search, it is easy to see that the `Search` procedure can be invoked from line 3 on page 38 and line 32 on page 39 a total of

[10] The curious reader is referred to section 5.1 of Appel [4] for implementation details.

```
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 5.5: SSI renaming algorithm, cont.

$O(E)$ 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.

We have shown that Algorithm 5.3 places all the required ϕ- and σ-functions in the control-flow graph according to SSI form conditions 1, 2, and 5; we will now show that this algorithm renames variables consistent with conditions 3 and 4 to prove that these algorithms combined suffice to convert a program into SSI form. The SSI form is not necessarily minimal, as we showed in section 5.3.3; the next section will show how to post-process to create minimal or pruned SSI form.

Lemma 5.3. *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 39 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`. $\square$

For brevity, we will hereafter refer to the canonical path constructed in the manner of lemma 5.3 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 $\mathrm{CP}(e)$ exists and is unique for every edge e in the CFG.

Lemma 5.4. *SSI form condition 3 (ϕ-function naming) holds for variables renamed according to Algorithm 5.4.*

Proof. We restate SSI form condition 3 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 $\mathrm{CP}(\langle Y', Y\rangle) = \mathtt{START} \xrightarrow{*} Y' \rightarrow Y$ for some use of a variable v at Y, constructed according to lemma 5.3 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 $\mathrm{CP}(e)$ is well defined and unique in the context of a use of v_i.

The canonical path $\mathtt{START} \xrightarrow{+} Y$ 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 $\mathrm{CP}(\langle Y', Y\rangle)$, for any use to receive a the

[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.

name ν. 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 $\mathtt{START} \xrightarrow{+} X \xrightarrow{+} Y$ exists; now we must prove that no other path from X to Y contains a definition of ν. Call this other definition D. Obviously D cannot be on our canonical path $\mathtt{START} \xrightarrow{+} X \xrightarrow{+} Y$, 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 ν leads to contradiction no other such path may exist, completing the proof of the lemma. □

Lemma 5.5. *SSI form condition 4 (σ-function naming) holds for variables renamed according to Algorithm 5.4.*

Proof. We restate SSI form condition 4 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 SSI condition 2 this node N requires a σ-function for V. If $N \neq Z$ then line 5 of Algorithm 5.4 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 condition 4 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 5.4 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 5.6. *SSI form condition 6 (correctness) holds for variables renamed according to Algorithm 5.4. 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 5.5 on page 39. 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 by the definition in section 3 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 5.5 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.

Case V: N_k has multiple predecessors and multiple successors. Forbidden by the CFG definition in section 3.

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 some variable V) follows. □

Theorem 5.2. *Algorithm 5.4 renames variables such that SSI form conditions 3, 4, and 6 hold.*

Proof. Direct from lemmas 5.4, 5.5, and 5.6. □

Theorem 5.3. *Algorithms 5.3 and 5.4 correctly transform a program into SSI form.*

Proof. Theorem 5.1 proves that ϕ- and σ-functions are placed correctly to satisfy conditions 1, 2 and 5 of the SSI form definition, and theorem 5.2 proves that variables are renamed correctly to satisfy conditions 3, 4 and 6.

□

5.3.6 Pruning SSI form

The SSI algorithm can be run using any conservative approximation to the liveness information (including the function $\texttt{MaybeLive}(v, n) = \texttt{true}$) if unused code elimination[12] is performed to remove extra ϕ- and σ-functions added and create pruned SSI. Figure 5.10 and Algorithm 5.6 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 $O(U_{SSI} + V_{SSI}) = O(U_{SSI})$.

5.3.7 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

[12] We follow [44] 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.

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 5.9: Environment datatype for the SSI renaming algorithm.

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 5.10: 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 5.6: Identifying unused code using SSI form.

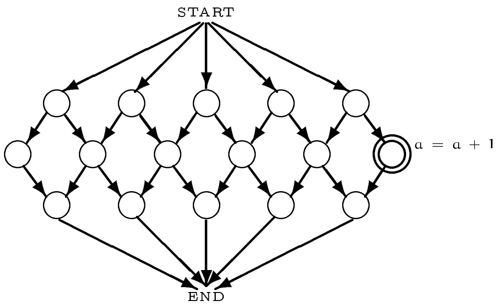


Figure 5.11: A worst-case CFG for "optimistic" algorithms.

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 my experience, optimistic algorithms tend to have poor time bounds because of the possibility of input graphs like the one illustrated in Figure 5.11. 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, an 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.

5.4 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 bounds are. A canonical example is [38], which states that for SSA form, "the number of ϕ-nodes needed remains linear." Typically Cytron [11] is cited; however, that reference actually reads:

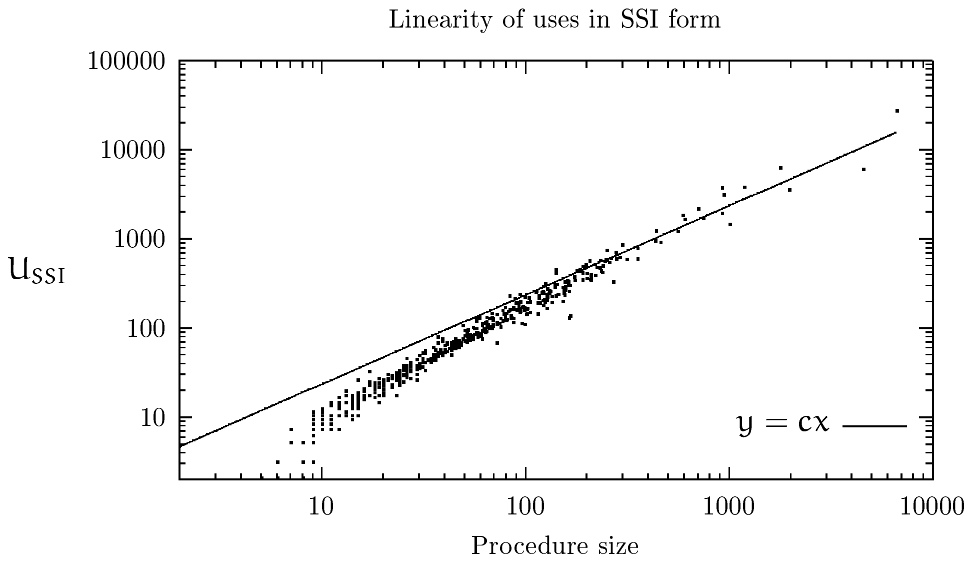


Figure 5.12: Number of uses in SSI form as a function of procedure length.

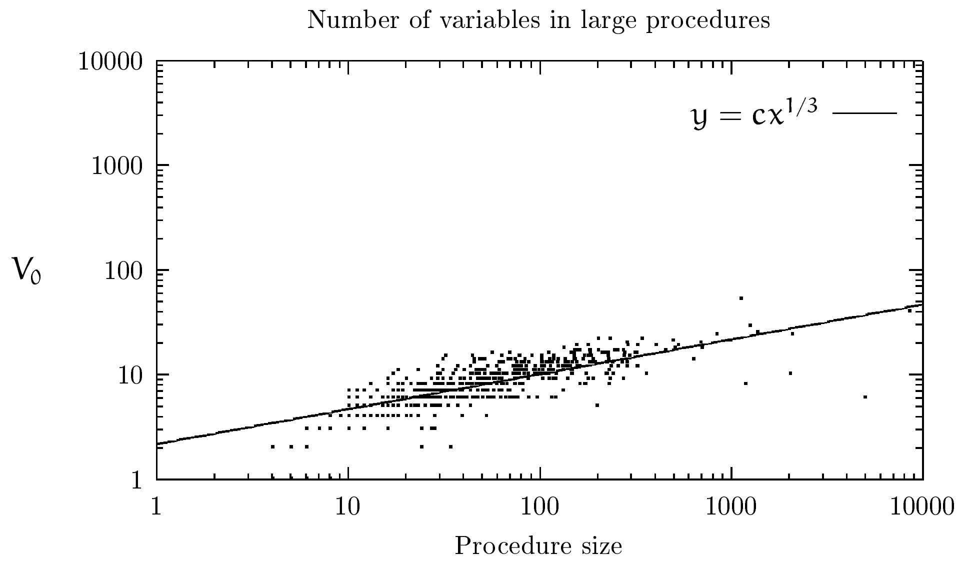


Figure 5.13: Number of original variables as a function of procedure length.

> 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 [23] 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 [44] 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 [44] 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.[13]

Thus the "linearity" of our SSI construction algorithm rests on the quantities NV_0 and U_{SSI}. Figures 5.12 and 5.13 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

[13] We will not follow Cytron [11] in defining a new variable R to denote $\max(N, E, \ldots)$ to avoid following him in declaring worst-case complexity $O(R^3)$ and leaving it to the reader to puzzle out whether $O(N^6)$ (!) is really being implied.

call-graph of the FLEX compiler itself, which includes large portions of the standard Java class libraries. Figure 5.12 shows convincingly that U_{SSI} grows as N for large procedures, and Figure 5.13 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 $O(E + V_{SSA}|DF| + NV_{SSA})$. 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[14] that the $|DF|$ term, the size of the dominance frontier, can be $O(N^2)$ for common programming constructs (`repeat-until` loops), which indicates that the $V_{SSA}|DF|$ 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 5.12 shows linear practical space use.

6 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

[14] See Dhamdhere [12] for example.

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.

6.1 Backward Dataflow Analysis

Backward dataflow analyses are those in which information is propagated in the direction opposite that of program execution [29]. There is general agreement [20, 7, 45] 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. Property 5.2 states that every non-ϕ-function use of a variable is dominated by the definition; Cytron [11] 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 [29]. 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 on the sparse SSI graph to the intermediate nodes on the control-flow graph. Formally, we can say that the dataflow property for variable ν at node N is dependent only on the properties at nodes D and U, defining and using ν, 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 6.1. *An expression* e *is* **very busy** *at a point* P *of the program iff it is always subsequently used before it is killed [29].*

Definition 6.2. *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 *[20].*

Johnson and Pingali [20] show how to reduce these properties of expressions to properties on variables. We will therefore consider properties $\mathrm{BSY}(\nu, N)$, $\mathrm{ANT}(\nu, N)$, and $\mathrm{PAN}(\nu, N)$ denoting very busy, totally anticipatable, and partially anticipatable variables ν 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:

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

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

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

The present section is concerned more with feasibility than the mechanics of implementation; we refer the interested reader to [29] and [20] 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 [22] 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.

6.2 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 [44] and removes unnecessary array-bounds and null-pointer checks, computes vari-

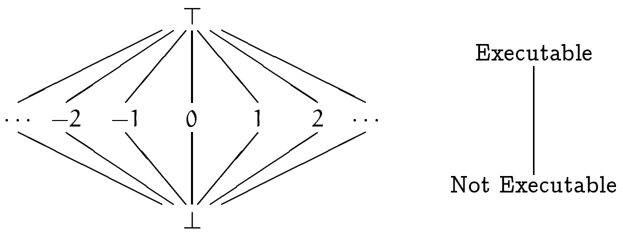


Figure 6.1: Three-level value lattice and two-level executability lattice for SCC.

$\sqcap$	$\bot$	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 6.1: Meet and binary operation rules on the SCC value lattice.

able types, and performs floating-point- and string-constant-propagation in addition to the integer constant propagation of standard SCC.

We will describe this algorithm incrementally, beginning with the standard SCC constant-propagation algorithm for review. Wegman and Zadeck's algorithm operates on a program in SSA form; we will call this SCC/SSA to differentiate it from SCC/SSI, using the SSI form, which we will describe in section 6.2.2. Section 6.3 on page 72 will discuss an extension to SPTC which does ***bit-width analysis***.

6.2.1 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 6.1.

```
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 6.1: SCC algorithm for SSA form.

RaiseE(e:edge) =

/* *When called,* $e \notin E_e$ */
$E_e \leftarrow E_e \cup \{e\}$
let n be the destination of edge e
if $n \notin E_n$ **then**
  $E_n \leftarrow E_n \cup \{n\}$
  $W_n \leftarrow W_n \cup \{n\}$

RaiseV(v:variable, L:lattice value) =

if $V[v] \sqsubset L$ **then**
  $V[v] \leftarrow L$
  $W_v \leftarrow W_v \cup \{v\}$

Visit(n:node) =

for each assignment "$v \leftarrow x \oplus y$" in n **do**
  RaiseV(v, $V[x] \oplus V[y]$) /* *binop rule: see table 6.1* */

for each assignment "$v \leftarrow$ MEM(...)" or "$v \leftarrow$ CALL(...)" in n **do**
  RaiseV(v, $\top$)

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, $V[v] \sqcap V[x_i]$) /* *meet rule: see table 6.1* */

for each branch "if v goto e_1 else e_2" in n **do**
  $L \leftarrow V[v]$
  if $L = \top$ or $L = c$ where c signifies "true" and $e_1 \notin E_e$ **then**
    RaiseE(e_1)
  if $L = \top$ or $L = c$ where c signifies "false" and $e_2 \notin E_e$ **then**
    RaiseE(e_2)

Algorithm 6.2: SCC algorithm for SSA form, cont.

Associating a lattice value with a definition point is a conservative statement that, for all possible program paths, the value of that variable has a certain property. The value lattice is, formally, $Int_{\perp}^{\top}$; the lattice value $\perp$ signifies that no information about the value is known, the lattice value $\top$ indicates that it is possible that the variable has more than one dynamic value, and the other lattice entries (corresponding to integer constants and occuping a flat space between $\top$ and $\perp$) indicate that the variable can be proven to have a single constant value in all runs of the program.[15] Similarly, the executability lattice indicates whether it is possible that the control flow edge is traversed in some execution of the program (marked "executable"), or if it can be proven that the edge is never traversed in any valid program path (marked "not executable"). The algorithm works with SSA form, and is presented as Algorithm 6.1. Binary operations on lattice values and combination at ϕ-nodes follow the rules in Table 6.1; notice that the meet operation ($\sqcap$) is simply the least upper bound on the lattice. The time complexity of SCC/SSA can be found easily: the procedure `RaiseE` puts each node on the W_n worklist at most once, and `RaiseV` puts a variable on the W_v worklist at most $D - 1$ times, where D is the maximum lattice depth. The `Visit` procedure can thus be invoked a maximum of N times by line 11 of the `Analyze` procedure of Algorithm 6.1, and a maximum of $U_{SSA}(D-1)$ times by line 15, where U_{SSA} is the number of variable ***uses*** in the SSA representation of the program. The lattice depth D is the constant 3 in this version of the algorithm, so it drops out of the expression. The `RaiseE` procedure itself is called at most E times. The time complexity is

[15] Note that we follow the $\top$ and $\perp$ conventions used in semantics and abstract interpretation; authors in dataflow analysis (including Wegman and Zadeck in their SCC paper [44]) often use contrary definitions, letting $\top$ mean undefined and $\perp$ indicate overdefinition. As section 7.3 will discuss the semantics of SSI^{+} at length, we thought it best to adhere to one set of definitions consistently, instead of switching mid-paper.

`foo = f();`	$\texttt{foo}_0$ `= f();`
`if (foo == 1)`	`if (`$\texttt{foo}_0$ `== 1)`
	$\langle \texttt{foo}_1, \texttt{foo}_2 \rangle = \sigma(\texttt{foo}_0)$
`    bar = foo + 1;`	$\quad\texttt{bar}_0 = \texttt{foo}_2 + 1;$
`else`	`else`
`    bar = 2;`	$\quad\texttt{bar}_1 = 2;$
	$\texttt{bar}_2 = \phi(\texttt{bar}_0, \texttt{bar}_1)$

Figure 6.2: A simple constant-propagation example.

thus $O(E + N + U_{SSA}(D - 1))$ which simplifies to $O(E + U_{SSA})$.

6.2.2 SCC/SSI: predication using σ-functions.

Porting the SCC algorithm from SSA to SSI form immediately increases the number of constants we can find. A simple example is shown in Figure 6.2: the version of the program on the right is in SSI form, and SCC/SSI—unlike SCC/SSA—can determine that $\texttt{foo}_2$ is a constant with value 1 (although nothing can be said about the value of $\texttt{foo}_0$ or $\texttt{foo}_1$) and therefore that $\texttt{bar}_0$, $\texttt{bar}_1$, and $\texttt{bar}_2$ are constants with the value 2. SSI form creates a new name for `bar` at the conditional branch to indicate that more information about its value is known.

Only the `Visit` procedure must be updated for SCC/SSI: lattice update rules for σ-functions must be added. Algorithm 6.3 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

```
Visit(n:node) =
/* Assignment rules as on page 58 */

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 6.3: A revised `Visit` procedure for SCC/SSI.

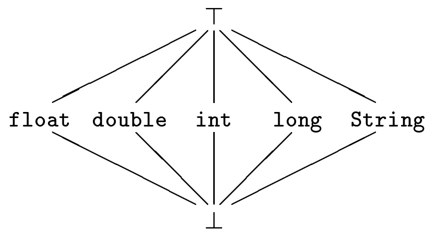


Figure 6.3: SCC value lattice extended to Java primitive value domain.

SCC/SSA: $O(E + U_{SSA})$, by the same argument as before.

6.2.3 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 [15]; the extended lattice is presented in Figure 6.3. 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.[16] 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 6.3, and will be omitted for brevity. In the next sections, the "int" and "long" entries in this lattice will be summarized as "Integer Con-

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

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 [28].

Table 6.2: Class hierarchy statistics for several large O-O projects.

stant", 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.

6.2.4 Type analysis

In Figure 6.4 we extend the lattice to compute Java type information. The new lattice entry marked "Typed" is actually forest-structured as shown in Figure 6.5; 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 6.4. 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 6.2 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.[17] 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.

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

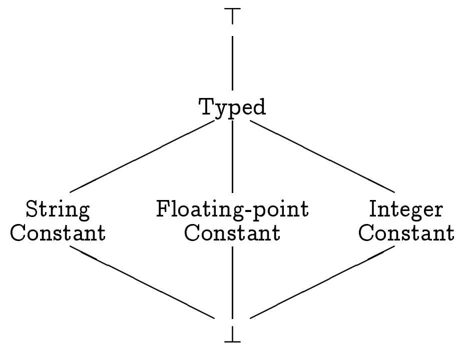


Figure 6.4: SCC value lattice extended with type information.

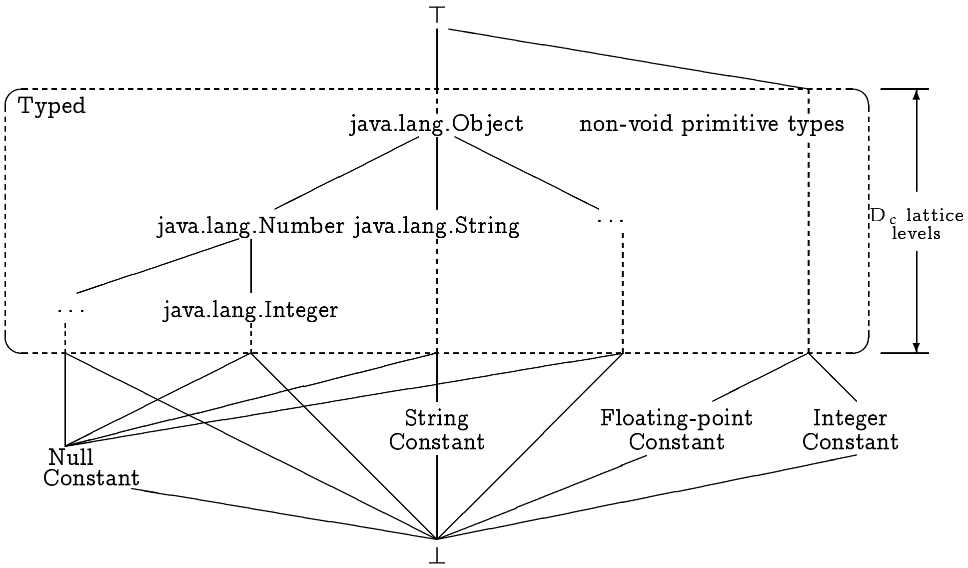


Figure 6.5: “Typed” category of Figure 6.4 shown expanded.

```
Visit(n:node) =
for each assignment "v ← x ⊕ y" in n do
  RaiseV(v, V[x] ⊕ V[y]) /* binop rule: see figure 6.6 */

for each assignment "v ← MEM(...)" or "v ← CALL(...)" in n do
  let t be the type of the MEM or CALL expression
  RaiseV(v, t)

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, ⊔_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 ⊑ L[x] or Typed ⊑ 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 "⟨v_1, v_2⟩ ← σ(v_0)" associated with this branch do
    if edge e_1 ∈ 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 ∈ E_e then
      RaiseV(v_1, L[v_0])
    if edge e_2 ∈ E_e then /* False branch */
      RaiseV(v_2, L[v_0])

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

Algorithm 6.4: `Visit` procedure for typed SCC/SSI.

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

Figure 6.6: Java typing rules for binary operations.

A brief word on the roots of the hierarchy forest in Figure 6.5 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`.[18] 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 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

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

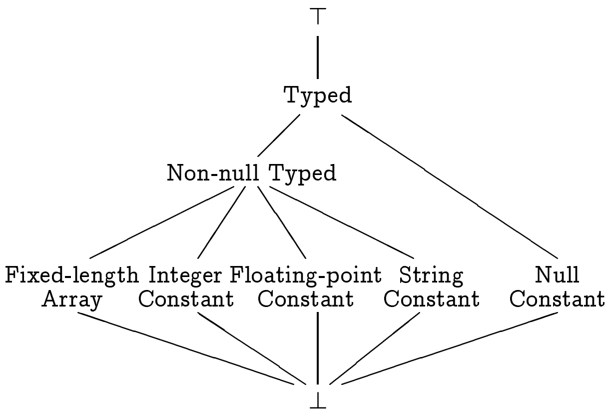


Figure 6.7: Value lattice extended with array and null information.

(which do not extend other interfaces) are additional roots in the hierarchy forest, although no examples of this are shown in Figure 6.5.

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 6.2.6 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.

6.2.5 Addressing array-bounds and null-pointer checks

At this point, we can expand the value lattice once more to allow elim-

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

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 6.8: Extended value lattice inequalities.

```
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 6.9: An example illustrating the power of combined analysis.

```
Visit(n:node) =
/* Binop and φ-function rules as in algorithm 6.4 */

for each assignment "v ← MEM(...)" or "v ← CALL(...)" in n do
   let t ∈ Class_possibly-null ∪ Class_primitive be the type of the MEM or CALL
   RaiseV(v, t)

for each array creation expression "v ← new T[x]" do
   if L[x] is an integer constant then
      RaiseV(v, A(T, L[x]))
   else
      RaiseV(v, T_non-null)

for each array length assignment "v ← 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 6.4, with the obvious extension to allow tests
   against null to lower a lattice value from Class_possibly-null to Class_non-null. */
```

Algorithm 6.5: `Visit` procedure outline with array and null information.

```
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 6.10: Implicit bounds checks (underlined) on Java array references.

ination of unnecessary array-bounds and null-pointer checks, based on our constant-propagation algorithm. The new lattice is shown in Figure 6.7; we have split the "Typed" lattice entry to enable the algorithm to distinguish between non-null and possibly-null values,[19] and added a lattice level for arrays of known constant length. Some formal definition of the new value lattice can be found in Figure 6.8; the meet rule is still the least upper bound on the lattice. Modifications to the `Visit` procedure are outlined in Algorithm 6.5. 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 6.9.

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

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

induction variable. In the example of Figure 6.10 on the preceding page, 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 preceding 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.[20] 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 6.11 on the facing page, 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 integer linear programming approach [36] to fully address array-bounds checks.

6.2.6 Experimental results

The full SPTC analysis and optimization has been implemented in the FLEX java compiler platform.[21] Some quantitative measure of the utility of SPTC is given as Figure 6.12. The "run-times" given are intermediate representation dynamic statement counts generated by the FLEX compiler SSI IR interpreter. The FLEX infrastructure is still under development, and its backends are not stable enough to allow directly executable code. As such, the numbers bear a tenuous relation to reality; in particular branch delays on real architectures, which the elimination of null-pointer checks seeks to eliminate, are unrepresented. Furthermore, the intermediate representation interpreter gives the same cycle-count to two-operand instructions as

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

[21] See section 8 for details of methodology.

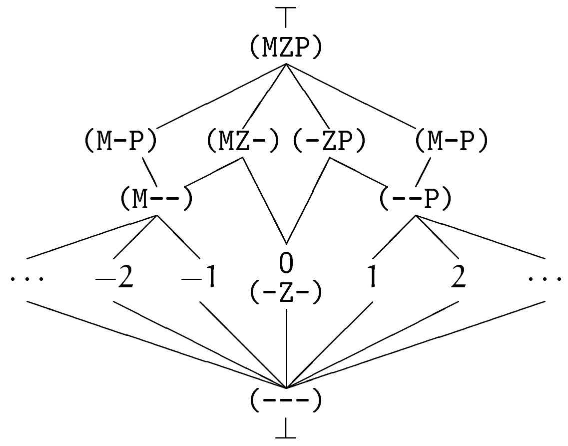


Figure 6.11: An integer lattice for signed integers. A classification into negative (M), positive (P), or zero (Z) is grafted onto the standard flat integer constant domain. The (M-P) entry is duplicated to aid clarity.

to loading constants, which tends to negate most of the benefit of constant propagation. As is obvious from the figure, 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. The "Hello world" 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.

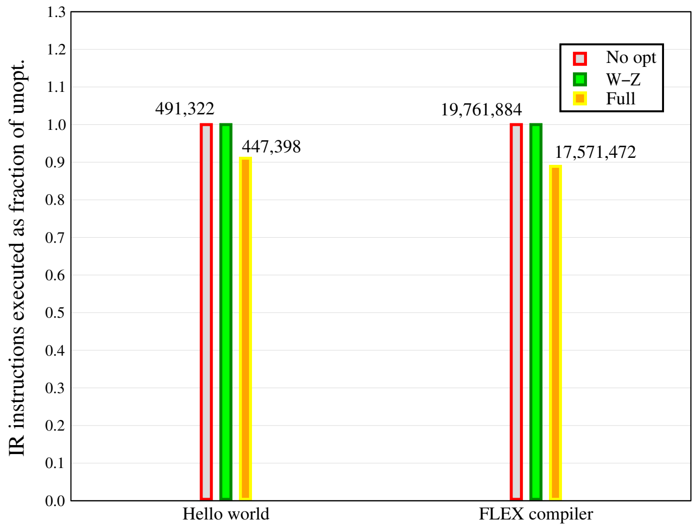


Figure 6.12: SPTC optimization performance.

6.3 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 [30]. To take advantage of these functional units without explicit human annotation, the compiler must be able to

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

Figure 6.13: Some combination rules for bit-width analysis.

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 [3] showed that 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 6.11 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 6.13 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 An executable representation

The Static Single Information (SSI) form, as presented in the first half of this thesis, requires control-flow graph information in order to be executable. We would like to have a demand-driven operational semantics for SSI form that does not require control-flow information; thus freeing us to more flexibly reorder execution.

In particular, we would like a representation that eliminates unnecessary control dependencies such as exist in the program of Figure 7.1 on the next page. A control-flow graph for this program, as it is written, will explicitly specify that no assignments to `B[]` will take place until all elements of `A[]` have been assigned; that is, the second loop will be *control-dependent* on the first. We would like to remove this control dependence in order to provide greater parallelism—in this case, to allow the assignments to `A[]` and `B[]` to take place in parallel, if possible.

In addition, an executable representation allows us to more easily apply the techniques of abstract interpretation [31]. Although abstract interpretation may be applied to the original SSI form using information extracted from the control flow graph, an executable SSI form allows more concise (and thus, more easily derived and verified) abstract interpretation algo-

```
for (int i=0; i<10; i++)
    A[i] = x;
for (int j=0; j<10; j++)
    B[j] = y;
```

Figure 7.1: An example of unnecessary control dependence: the second loop is *control-dependent* on the first and so assignments to `A[]` and `B[]` cannot take place in parallel.

rithms.

The modifications outlined here extend SSI form to provide a useful and descriptive operational semantics. We will call the extended form SSI^+. For clarity, SSI form as originally presented we will call SSI_0. We will describe algorithms to contruct SSI^+ efficiently, and illustrate analyses and optimizations using the form.

7.1 Deficiencies in SSI_0

Although a demand-driven execution model can be constructed for SSI_0, it fails to handle loops and imperative constructs well. SSI^+ form addresses these deficiencies.

7.1.1 Imperative constructs, pointer variables, and side-effects

The presentation of SSI_0 ignored pointers, concentrating on so-called register variables. Extending SSI_0 to handle these imperative constructs is quite easy: we simply define a "variable" S to represent an updatable store. This variable is renamed and numbered as before, so that S_0 represents the initial contents of the store and $S_i, i > 0$ represents the contents of the store after some sequence of writes. Figure 7.2 shows a simple imperative program in

```
// swap A[i] and B[j]        // SSI+ form:
x = A[i];                    x0 = FETCH(S0, A0 + i0)
y = B[j];                    y0 = FETCH(S0, B0 + j0)
A[i] = y;                    S1 = STORE(S0, A0 + i0, y0);
B[j] = x;                    S2 = STORE(S1, B0 + j0, x0);
```

Figure 7.2: Use of the "store variable" S_x in SSI^+ form.

SSI^+ form. Note that modifications to the store typically take the previous contents of the store as input, and that subroutines with side-effects modifying the store must be written in SSI^+ form such that they both take a store and return a store.

The single monolithic store may provide aliasing at too coarse a resolution to be useful. Decomposing the store into smaller regions is a straightforward application of pointer analysis, which may benefit from an initial conversion of register variables to SSI_0 form. In type-safe languages, defining multiple stores for differing type sets is a trivial implementation of basic pointer analysis; Figure 7.3 shows a simple example of this form of decomposition using two different subtypes (`Integer` and `Float`) of a common base class (`Number`). Pointer analysis is a huge and rapidly-growing field which we cannot attempt to summarize here; suffice to say that the *may-point-to* relation from pointer analysis may be used to define a fine-grained model of the store.

Proper sequencing among statements with side-effects may be handled in a similar way: a special SSI name is used/defined where side-effects occur to impose an implicit ordering. For maximum symmetry with the 'store' case, we will name this special variable S^{fx}. This variable may be further decomposed using effect analysis for more precision.

Note that precise analysis of side-effects and the store is much more

$$\vdash \mathsf{N} : \textbf{Number}, \mathsf{I} : \textbf{Integer}, \mathsf{F} : \textbf{Float}$$

$$\mathsf{I} \subset \mathsf{N} \text{ and } \mathsf{F} \subset \mathsf{N}$$

`if(P)`	*// SSI$^+$ form:*
`    N=I;`	
`else`	$N_0 = \phi(I_0, F_0)$
`    N=F;`	
`F.add(3.14159);`	$S^F_1 = \mathtt{CALL}(\mathsf{add}, S^F_0, F_0, 3.14159)$
`N.add(5);`	$\left\langle S^I_1, S^F_2 \right\rangle = \mathtt{CALL}(\mathsf{add}, S^I_0, S^F_1, N_0, 5)$

Figure 7.3: Factoring the store (S_x) using type information in a type-safe language.

important in C-like languages. The example on the left in Figure 7.4 shows the difficulties one may encounter in dealing with pointer variables that may rewrite SSI temporaries. It is possible to deal with this in the manner of Figure 7.3 using explicit stores, and with sufficient analysis one may write the SSI representation on the right in the figure. The source language for our FLEX compiler does not encounter this difficulty: Java has no pointers to base types, and so the compiler does not have to worry about values changing "behind its back" as in the example.

7.1.2 Loop constructs

The center column of Figure 7.5 on page 79 shows a typical loop in SSI_0 form. Note first that an explicit "control flow" expression (`goto L1`) is required in order to make sense of the program. Note also that i_1, i_2 and i_3 are potentially *dynamically* assigned many times, although *statically* they have only one definition each. This complicates any sort of demand-driven semantics: should the ϕ-function demand the value of i_0, or i_3,

`int x=1;`	$x_0 = 1$
`int y=2;`	$y_0 = 2$
`int *p = &x;`	$p_0 = \{x\}$ *// P is of type "location set"*
`if (P)`	
`    p = &y;`	$p_1 = \{y\}$
	$p_2 = \phi(p_0, p_1)$
`*p = 3;`	$\langle x_1, y_1 \rangle = \mathrm{DEREF}(p_2, 3)$
`return x;`	`return` x_1

Figure 7.4: Pointer manipulation of local variables in C.

when it is evaluated the first time? Which of the values of i_3 does it receive when the ϕ-function is subsequently evaluated? A token-based dataflow interpretation fails as well: it is easy to see that tokens for i_x flow around the loop before flowing out at the end, but the token for j_0 seems to be "used up" in the first iteration.

SSI$^+$ introduces a ξ-function in the block of ϕ-functions to clarify the loop semantics. The left-hand column of Figure 7.5 illustrates the nature of this function. The ξ-function arbitrates loop iteration, and will be defined precisely by the operational semantics of SSI$^+$ form. For now note that it relates iteration variables (the top tuple of the parameter and result vectors) to loop invariants (the bottom tuple of the vectors). We followed the statement ordering of SSI$_0$ in the figure, but unlike SSI$_0$, the statements of SSI$^+$ could appear in any order without affecting their meaning—and so the statement label `L1` of the SSI$_0$ representation and its implicit control-flow edge are unnecessary in SSI$^+$.

// a simple loop	*// SSI_0 form:*	*// SSI^+ form:*
`j=1;`	$j_0 = 1$	$j_0 = 1$
`i=0;`	$i_0 = 0$	$i_0 = 0$
`do`	`L1:`	$\begin{bmatrix}\langle j_1\rangle \\ \langle i_5\rangle\end{bmatrix} = \xi(\begin{bmatrix}\langle j_0\rangle \\ \langle i_3\rangle\end{bmatrix})$
`{`	$i_1 = \phi(i_0, i_3)$	$i_1 = \phi(i_0, i_5)$
`    i+=j;`	$i_2 = i_1 + j_0$	$i_2 = i_1 + j_1$
`} while (i<5);`	$P_0 = (i_2 < 5)$	$P_0 = (i_2 < 5)$
	`if` P_0 `goto L1`	$\langle i_3, i_4\rangle = \sigma(P_0, i_2)$
	$\langle i_3, i_4\rangle = \sigma(i_2)$	

Figure 7.5: A simple loop, in SSI_0 and SSI^+ forms.

7.2 Definitions

The signature characteristic of SSI^+ are the ξ-functions. These ξ-functions exist in the same places ϕ-functions do, and control loop iteration. The exact semantics may vary—the sections below present two different valid semantics for a ξ-functions—but informally they can be viewed as "time-warp" operators. They take values from the "past" (previous iterations of the loop or loop invariants valid when the loop began) and project them into the "future" (the current loop iteration).

There is at most one ξ-function per ϕ-function block, and it always precedes the ϕ-functions. Construction of ξ-functions takes place before the renaming step associated with SSI form, and the ξ-functions are then renamed in the same manner as any other definition. The top tuple of the constructed ξ-function contains the names of all variables reaching the guarded ϕ-function via a backedge, and the bottom tuple contains all variables used inside the guarded loop that are *not* mentioned in the header's ϕ-function.

The SSI$^+$ form also has ***triggered constants***. The time-oriented semantics of SSI$^+$ dictate that each constant must be associated with a trigger specifying for what times (cycles/loop iterations) the value of the constant is valid. These are similar to the constant generators in some dataflow machines [42]. The triggers for a constant c come from the variables defined in the earliest applicable instruction post-dominated by the constant definition statement $\nu = c$. This is designed to generate the trigger as soon as it is known that the constant definition statement will always execute. In practice it is necessary to introduce a bogus ***trigger variable***, $\mathcal{C}_T$ which is generated at the START node and used to trigger any constants otherwise without a suitable generator. If the use of the constant does not post-dominate the START node, $\mathcal{C}_T$ will have to be threaded through ϕ- and σ-functions to reach the earliest post-dominated node.

7.3 Semantics

We will base the operational semantics of SSI$^+$ on a demand-driven dataflow model. We will define both a cycle-oriented semantics and an event-driven semantics, which (incidentally) correspond to synchronous and asynchronous hardware models.

Following the lead of Pingali [31], we present Plotkin-style semantics [33] in which *configurations* are rewritten instead of programs. The configurations represent program state and transitions correspond to steps in program execution. The set of valid transitions is generated from the program text.

The semantics operate over a lifted value domain $V = Int_\perp$. When some variable $t = \perp_V$ we say it is *undefined*; conversely $t \sqsupset \perp_V$ indicates that the variable is *defined*. "Store" metavariables S_x are not explicitly handled by the semantics, but the extension is trivial with an appropriate redefinition of the value domain V. Floating-point and other types are also trivial extensions. The metavariables c and ν stand for elements of V.

We also define a domain of *variable names*, $Nam = \{n_0, n_1, \ldots\}$. The metavariables t and P stand for elements in *Nam*, although P will be reserved for naming branch predicates.

A fixed set of "built-in" operators, **op**, is defined, of type $V^* \to V$. If any operator argument is $\perp$, the result is also $\perp$. Constants are implemented as a special case of the general operator rule: an **op** producing a constant has a single trigger input which does not affect the output.

7.3.1 Cycle-oriented semantics

In the cycle-oriented semantics, configurations consist of an *environment*, ρ, which maps names in *Nam* to values in V.

Definition 7.1.

1. *An* environment $\rho : \mathrm{N} \to \mathrm{V}$ *is a finite function—its domain* $\mathrm{N} \subseteq \mathrm{Nam}$ *is finite. The notation* $\rho[t \mapsto c]$ *represents an environment identical to* ρ *except for name* t *which is mapped to* c*.*

2. *The null environment* $\rho_\emptyset$ *maps every* $t \in \mathrm{N}$ *to* $\perp_\mathrm{V}$*.*

3. *A* configuration *consists of an environment. The initial configuration is* $\rho_\emptyset[\mathcal{C}_\mathrm{T} \to 0]$ *extended with mappings for procedure parameters. That is, all names in* N *are mapped to* $\perp_\mathrm{V}$ *except for the default constant trigger* C_T *mapped to 0,*[22] *and any procedure parameters mapped to their proper entry values.*

Figure 7.6 on the next page shows the cycle-oriented transition rules for SSI$^+$ form. The left column consists of definitions and the right column

[22] Any $k \sqsupset \perp_V$ would do.

$$t = \mathbf{op}(t_1, \ldots, t_n) : \qquad \frac{\rho[t] = \bot \wedge (\rho[t_1] \sqsupset \bot \wedge \ldots \wedge \rho[t_n] \sqsupset \bot)}{\rho \rightarrow \rho[t \mapsto \mathbf{op}(\rho[t_1], \ldots, \rho[t_n])]}$$

$$t = \phi(t_1, \ldots, t_n) : \qquad \frac{\rho[t] = \bot \wedge \rho[t_j] \sqsupset \bot \wedge \text{all other } \rho[t_1], \ldots, \rho[t_n] = \bot}{\rho \rightarrow \rho[t \mapsto \rho[t_j]]}$$

$$\langle t_1, \ldots, t_n \rangle = \sigma(P, t) : \qquad \frac{\rho[P] = \nu \sqsupset \bot \wedge \rho[t_{\nu-1}] = \bot \wedge \rho[t] \sqsupset \bot}{\rho \rightarrow \rho[t_{\nu-1} \mapsto \rho[t]]} \quad \text{where } (0 \leq \nu \leq n-1)$$

$$\left[\begin{smallmatrix}\langle t_1,\ldots,t_n\rangle \\ \langle t_{n+1},\ldots,t_m\rangle\end{smallmatrix}\right] = \xi(\left[\begin{smallmatrix}\langle t'_1,\ldots,t'_n\rangle \\ \langle t'_{n+1},\ldots,t'_m\rangle\end{smallmatrix}\right]) : \qquad \frac{\rho[t_j] = \bot \wedge \rho[t'_j] \sqsupset \bot}{\rho \rightarrow \rho[t_j \mapsto \rho[t'_j]]} \quad \text{where } (1 \leq j \leq n)$$

$$\left[\begin{smallmatrix}\langle t_1,\ldots,t_n\rangle \\ \langle t_{n+1},\ldots,t_m\rangle\end{smallmatrix}\right] = \xi(\left[\begin{smallmatrix}\langle t'_1,\ldots,t'_n\rangle \\ \langle t'_{n+1},\ldots,t'_m\rangle\end{smallmatrix}\right]) : \qquad \frac{\rho[t'_{n+1}] \sqsupset \bot \wedge \ldots \wedge \rho[t'_m] \sqsupset \bot}{\begin{array}{l}\rho \rightarrow \rho_\emptyset[t_1 \mapsto \rho[t_1]] \ldots [t_n \mapsto \rho[t_n]] \\ \qquad [t_{n+1} \mapsto \rho[t'_{n+1}]] \ldots [t_m \mapsto \rho[t'_m]]\end{array}}$$

Figure 7.6: Cycle-oriented transition rules for SSI$^+$.

shows a precondition on top of the line, and a transition below the line. If the definition in the left column is present in the SSI$^+$ form and the precondition on top of the line is satisfied, then the transition shown below the line can be performed.

7.3.2 Event-driven semantics

In the event-driven semantics, configurations consist of an *event set* and an *invariant store*. The event set E contains definitions of the form $t = c$, and the invariant store is a mapping from numbered ξ-functions in the source SSI$^+$ form to a set of tuples representing saved values for loop invariants.

We define the following domains:

$$t = \mathbf{op}(t_1, \ldots, t_n): \qquad \langle E[t_1 = v_1] \ldots [t_n = v_n], S\rangle \to \langle E[t = \mathbf{op}(v_1, \ldots, v_n)], S\rangle$$

$$t = \phi(t_1, \ldots, t_n): \qquad \langle E[t_i = v], S\rangle \to \langle E[t = v], S\rangle$$

$$\langle t_1, \ldots, t_n\rangle = \sigma(P, t): \qquad \langle E[t = v][P = i], S\rangle \to \langle E[t_i = v], S\rangle$$

$$\left[\begin{smallmatrix}\langle t_1,\ldots,t_n\rangle \\ \langle t_{n+1},\ldots,t_m\rangle\end{smallmatrix}\right] = \xi_K(\left[\begin{smallmatrix}\langle t'_1,\ldots,t'_n\rangle \\ \langle t'_{n+1},\ldots,t'_m\rangle\end{smallmatrix}\right]): \qquad \begin{array}{l}\langle E[t'_i = v], S\rangle \to \\ \qquad \langle E[t_i = v], S[K \mapsto S[K] \cup \langle t_i, v\rangle]\rangle \\ \qquad\qquad \text{where } 1 \le i \le n\end{array}$$

$$\left[\begin{smallmatrix}\langle t_1,\ldots,t_n\rangle \\ \langle t_{n+1},\ldots,t_m\rangle\end{smallmatrix}\right] = \xi_K(\left[\begin{smallmatrix}\langle t'_1,\ldots,t'_n\rangle \\ \langle t'_{n+1},\ldots,t'_m\rangle\end{smallmatrix}\right]): \qquad \frac{S[K] = \{\langle t_1, v_1\rangle, \ldots, \langle t_n, v_n\rangle\}}{\begin{array}{l}\langle E[t'_{n+1} = v_{n+1}] \ldots [t'_m = v_m], S\rangle \to \\ \qquad \langle E[t_1 = v_1] \ldots [t_m = v_m], S\rangle\end{array}}$$

Figure 7.7: Event-driven transition rules for SSI$^+$. In the last two rules K is a statement-identifier constant which is unique for each source ξ-function.

- *Evt* = *Nam* × *V* is the event domain. An event consists of a name-value pair. The metavariable e stands for elements of *Evt*.
- *Xif* ⊂ *Int* is used to number ξ-functions in the source SSI$^+$ form. There is some mapping function which relates ξ-functions to unique elements of *Xif*. The metavariable K stands for an element in *Xif*.

A formal definition of our configuration domain is now possible:

Definition 7.2.

1. *An* event set $E : \mathrm{Evt}^*$. *The notation* $E[t = c]$ *represents an event set identical to* E *except that it contains the event* $\langle t, c\rangle$. *We say a name* t *is* defined *if* $\langle t, v\rangle \in E$ *for some* v. *For all* $\langle t_1, v_1\rangle, \langle t_2, v_2\rangle \in E$, t_1 *and* t_2 *differ. This is equivalent to saying that no name* t *is*

multiply defined in an event set. This constraint is enforced by the transition rules, not by the definition of E*.*

2. *An* invariant store $S : \mathrm{Xif} \rightarrow \mathrm{Evt}^*$ *is a finite mapping from* ξ*-functions to event sets.*

3. *A* configuration *is a tuple* $\langle E, S\rangle : \mathrm{Evt}^* \times (\mathrm{Xif} \rightarrow \mathrm{Evt}^*)$ *consisting of an event set and an invariant store. The initial configuration for procedure parameters* $p_0, \ldots, p_n$ *mapped to non-*$\perp$ *values* $v_0, \ldots, v_n$ *is* $\langle\{\mathcal{C}_T = 0, p_0 = v_0, \ldots, p_n = v_n\}_{\mathrm{Evt}}, []_{\mathrm{Xif}\rightarrow\mathrm{Evt}^*}\rangle$ *that is, it consists of an empty event set extended with events for default constant trigger* $\mathcal{C}_T$ *and the procedure parameters, and an empty mapping for the invariant store.*

Figure 7.7 on the preceding page shows the event-driven transition rules for SSI^+ form. As before, the left column consists of definitions and the right column shows an optional precondition above a line, and a transition. If the definition in the left column is present in the SSI^+ form and the precondition (if any) above the line is satisfied, then the transition can be performed. Note that most transitions remove some event from the event set E, replacing it with a new event. The invariant store S stores the values of loop invariants for regeneration at each loop iteration.

7.4 Construction

Construction of SSI^+ is only a slight variation on the construction algorithms for SSI_0. First, dominator and post-dominator trees are produced using the Lengauer-Tarjan [25] or Harel [16] algorithm. The nodes of the dominator tree are numbered in pre-order such that for all nodes N, $\mathrm{num}[N] > \mathrm{num}[\mathrm{idom}[N]]$. Then, in a single traversal of the post-dominator tree, we find the lowest-numbered node post-dominated by any given node.

We add triggers to constants from variables defined at this lowest node post-dominated by the constant use; using the default trigger $\mathcal{C}_\mathrm{T}$ where necessary. We then place ϕ- and σ-functions for all variables, including constant triggers, using Algorithm 5.3.

We then generate ξ-functions. A standard interval analysis creates a loop nesting tree, and each loop is scanned for invariants and other definitions/uses to create the proper ξ-function tuples. Renaming is done using Algorithm 5.4, as before.

7.5 Dataflow and control dependence

The SSI$^+$ semantics are data-driven, and thus bring to mind work on compilers for dataflow machines. Beck, Johnson, and Pingali have previously written [6] on the benefits of dataflow-oriented intermediate representations. However, the previous work on dataflow compilers (Traub [42], for example) has concentrated on intra-loop dependencies, often leaving in pseudo-control-flow edges to serialize non-loop structures. This strategy results in the sort of fine-grain intra-loop parallelism suitable for parallel dataflow machines, vector processors, and VLIW machines.

The current work concentrates on removing unnecessary dependencies *between* loops, which allows a coarser parallelism which does not require as many functional units to take advantage of. Moreover, we extract parallel sequential threads that are not loop-based. Obviously both fine-grain and coarse-grain parallelism are important, but we feel the current industry trends towards loosely coupled multiprocessors support our coarser-grained approach which has, to date, seemingly been neglected by dataflow approaches.

7.6 Hardware compilation.

The observant reader may have noticed that the two operational semantics given in section 7.3 closely resemble circuit implementations for the program according to synchronous and asynchronous design methodologies. In fact, SSI^{+} was designed specifically to facilitate rendering a high-level program into hardware. The two semantics differ primarily on how cyclic dependencies (i.e. loops) are handled.

Translation of high-level languages directly to hardware has long been a goal of researchers. Tanaka et al. constructed a system based on FORTRAN [41], and Galloway's C-based hardware description language [13] inspired a new interest in applying general-purpose languages to the task. The recent general use of type-safe object-oriented languages has encouraged speculation that the more favorable analysis properties of these stricter languages would enable further advances in general-use hardware compilation. In this context, the well-defined semantics and data-flow orientation of SSI^{+} solve the local-level hardware compilation problem and allow effort to be concentrated on the more difficult intra-procedural analyses required.

8 Methodology

The SSI intermediate representation described in this paper is the core IR for the FLEX compiler infrastructure project, started in July 1998 and currently containing about 70,000 lines of Java source code. The FLEX compiler reads in Java bytecodes, and targets both the JVM (for high-level portable code transformations) and several combinations of machine architectures and runtime systems. Currently the bytecode and ARM processor backends are near completion. Interpreters exist for the various intermediate representations used in the compiler, allowing the correctness of the earlier passes of the compiler to be verified. The compiler will correctly compile itself to IR and interpret itself.

The FLEX compiler implements the algorithms described in this paper, validating their correctness. Variable counting for the graphs of section 5.4 was done by a special statistics module that could be applied to the results of any pass. The full bitwidth-extended SPTC constant propagation algorithm was implemented, although we currently do not use the bitwidth information produced. SSI^+ and hardware compilation are the focus of current work.

9 Conclusions

The Static Single Information form extends SSA without adding unneeded complexity to allow efficient predicated analysis and backward dataflow analyses. Futher, the SSI^+ variant removes all explicit control-dependence relations, allowing extraction of parallelism from the code, and possesses a complete and straight-forward semantics which makes it useful for, among other things, abstract interpretation and hardware compilation.

We have demonstrated efficient construction of SSI form, and several optimizations which use it to obtain efficiency improvements over previous methods. The many SSA-variant papers in the literature attest to limitations of standard SSA form; we believe SSI form solves these problems in a simple and symmetric manner. The FLEX compiler infrastructure demonstrates the practicality of SSI form.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988. The "Dragon Book".

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–11, San Diego, California, Jan. 1988.

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

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

[5] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, White Plains, New York, June 1990.

[6] M. Beck, R. Johnson, and K. Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, June 1991.

[7] J.-D. Choi, R. Cytron, and J. 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, Jan. 1991.

[8] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 273–286, Las Vegas, Nevada, May 1997.

[9] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the Sixth International Conference on Compiler Construction*, pages 253–267, Apr. 1996.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. 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, Jan. 1989.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[12] D. M. Dhamdhere, B. K. Rosen, and F. K. 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.

[13] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines. Proceedings*, pages 136–144, Apr. 1995.

[14] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.

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

[16] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[17] P. Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Rice University, Houston, Texas, May 1994.

[18] R. Johnson, D. Pearson, and K. Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report TR 93-1365, Cornell University, Ithaca, NY 14853-7501, July 1993.

[19] R. Johnson, D. Pearson, and K. 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.

[20] R. Johnson and K. 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.

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

[22] R. Kennedy, F. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. In *Proceedings of the Seventh International Conference on Compiler Construction*, pages 144–158, Lisbon, Portugal, Apr. 1998.

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

[24] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 1–12, Atlanta, Georgia, May 1999.

[25] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[26] S.-W. Liao, A. Diwan, R. P. B. Jr., A. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 37–48, Atlanta, Georgia, May 1999.

[27] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, Montreal, Canada, June 1998.

[28] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *Proceed-*

ings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pages 42–51, Atlanta, Georgia, Jan. 1996.

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

[30] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, Jan. 1997.

[31] K. Pingali, M. Beck, R. C. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report TR 90-1152, Cornell University, Ithaca, NY 14853-7501, Sept. 1990.

[32] K. Pingali and G. Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[33] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[34] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, Feb. 1981.

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL)*, pages 12–27, San Diego, California, Jan. 1988.

[36] R. Rugină and M. Rinard. Automatic parallelization of divide and conquer algorithms. Slides for talk given at PPoPP '99; available

from `http://www.cag.lcs.mit.edu/~rinard/divide_and_conquer/ppopp99.slides.ps`, May 1999.

[37] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, Feb. 1970.

[38] V. C. Sreedhar and G. 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, Jan. 1995.

[39] V. C. Sreedhar, G. R. Gao, and Y. Lee. A new framework for exhaustive and incremental data flow analysis using DJ graphs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–290, Philadelphia, Pennsylvania, May 1996.

[40] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: a proposed solution. *Communications of the ACM*, 1(8):12–18, Aug. 1958. Part 2: 1(9):9-15. Report of the Share Ad-Hoc committee on Universal Languages.

[41] T. Tanaka, T. Kobayashi, and O. Karatsu. HARP: FORTRAN to silicon. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):649–660, June 1989.

[42] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT/LCS/TR-370, Massachusetts Institute of Technology, Cambridge, MA 02139, Aug. 1986.

[43] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–55, La Jolla, California, June 1995.

[44] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[45] D. Weise, R. F. Crew, M. Ernst, and B. 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, Jan. 1994.