SSI revisited: A Program Representation for Sparse Data-flow Analyses

Andre Tavares[a], Mariza Bigonha[a], Roberto S. Bigonha[a], Benoit Boissinot[b], Fernando M. Q. Pereira[a], Fabrice Rastello[b]

[a] *UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil*
[b] *ENS Lyon – 46 allée d'Italie, 69364 Lyon, France*

Abstract

Data-flow analyses usually associate information about variables with program regions. Informally, if these regions are too small, e.g., a point between two consecutive statements, we call the analysis dense. On the other hand, if these regions include many such points, then we call it sparse. This paper presents a systematic method to build program representations that support forward and/or backward sparse analyses. To pave the way that leads to this framework we survey and clarify the bibliography about intermediate program representations. We revisit the Static Single Information (SSI) form introduced in the nineties and show how to simplify the construction of program representations for unidirectional data-flow analyses. We show that our approach, up to parameter choice, subsumes other program representations such as the SSA, SSI and e-SSA forms. For data-flow problems that can be partitioned by variables (PVP) we can produce intermediate representations isomorphic to Choi *et al.*'s Sparse Evaluation Graphs (SEG). However, contrary to SEGs, we can handle - sparsely - problems that are not PVP. We have implemented this framework in the LLVM compiler, and have empirically compared different program representations in terms of size and construction time.

Keywords: Compiler, Program representation, Data-flow analysis,

Email addresses: andrelct@dcc.ufmg.br (Andre Tavares), mariza@dcc.ufmg.br (Mariza Bigonha), bigonha@dcc.ufmg.br (Roberto S. Bigonha), benoit.boissinot@ens-lyon.fr (Benoit Boissinot), fernando@dcc.ufmg.br (Fernando M. Q. Pereira), fabrice.rastello@ens-lyon.fr (Fabrice Rastello)

1. Introduction

The monotone data-flow framework is an old ally of compiler writers. Since the work of pionners like Prosser [1], Allen [2, 3], Kildall [4] and Hecht [5], data-flow analyses such as reaching definitions, available expressions and liveness analysis have made their way into the implementation of virtually every important compiler. The information acquired by data-flow analyses supports many classic compiler optimizations, such as common-subexpression and dead-code elimination, constant and copy propagation, register allocation and pointer analysis, among others. Furthermore, this framework provides a core theory grounding a profusion of developments in compiler research, both in the academia and in the industry.

The vast majority of data-flow analyses binds information to pairs formed by a variable and a program point [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. For instance, for each program point p, and each integer variable v live at p, Stephenson *et al.*'s [18] bit-width analysis finds the size, in bits, of v at p. Although well studied in the literature, this approach has some drawbacks; in particular, it suffers from an excess of redundant information. For instance, a given variable v may be mapped to the same bit-width along many consecutive program points. Therefore, a natural way to reduce redundancies is to make these analyses *sparser*, increasing the granularity of the program regions that they manipulate. We identify two main design strategies to achieve this sparsity: the use of new data-structures that represent the program under analysis, or the use of new program representations which make it natural to associate information to larger code regions.

In terms of data-structures, the first, and best known method proposed to support sparse data-flow analyses is Choi *et al.*'s *Sparse Evaluation Graph* (SEG) [21]. The nodes of this graph represent program regions where information produced by the data-flow analysis might change. Choi *et al.*'s ideas have been further expanded, for example, by Johnson *et al.*'s *Quick Propagation Graphs* [11], or Ramalingan's *Compact Evaluation Graphs* [22]. Nowadays we have efficient algorithms that build such data-structures [23, 24, 25]. These data-structures improve many data-flow analyses in terms of runtime and memory consumption. Nevertheless, the elegance of SEGs and its successors have not, so far, been enough to attract the attention of mainstream compiler writers. Compilers such as gcc, LLVM or Java Hotspot rely, instead, on several types of program representations to provide support to sparse data-flow analyses.

The most famous among these representations is the Static Single Assignment form [26], which suits well forward flow analyses, such as reaching definitions. Other representations, not as popular, yet more general than SSA form, exist too. For instance, Scott Ananian has introduced in the late nineties the *Static Single Information* (SSI) form, a program representation that supports both forward and backward analyses [27]. This representation was later discussed by Jeremy Singer [28] and revisited by Boissinot *et al.* [29]. Singer provided new algorithms plus examples of applications that benefit from the SSI form, and Boissinot *et al.* clarified a number of omissions in the related literature. A different program representation – the *Extended Static Single Assignment* (e-SSA) form – was introduced by Bodik *et al.* [6]. As opposed to SSI and SSA, the e-SSA form supports flow analyses that obtain information both from variable definitions and conditional tests. Another important representation, which supports data-flow analyses that acquire information from uses, is the *Static Single Use* form (SSU). As uses and definitions are not fully symmetric (the live-range can "traverse" a use while it cannot traverse a definition) there exists different variants of SSU (eg. [14, 30, 31]). For instance, the "strict" SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. All these program representations are very effective, having seen use in a number of implementations of flow analyses; however, they only fit specific data-flow problems.

In this paper we present a method to build program representations that support sparse data-flow analyses. We build these program representations by splitting the live ranges of variables, in such a way that the information associated with variables is invariant along their entire live ranges. Our technique is more general than the program representations that we have mentioned before. It can be parametrized according to the direction(s) of the flow problems, i.e., forward and/or backward, and according to the program points where data-flow information is produced. Usually these points contain variable definitions, uses or conditional tests. In order to build these program representations, we use an algorithm that is as powerful as the method that Singer has used to convert a program to the SSI form [28]. However, our algorithm is simpler: as we show in Section 3, for all unidirectional and all non-truly bidirectional data-flow analysis we can avoid iterating the live range splitting process in order to build intermediate representations.

Our method subsumes Choi *et al.*'s sparse evaluation graphs, as we demonstrate in Appendix A; however, we improve on SEGS in a number of ways.

Firstly, SEGs best suit a class of data-flow analyses that Zadeck defines as *Partitioned Variable Problems* [32] (PVP). Reaching definitions and liveness analysis are examples of PVPs. For these problems we can build intermediate program representations isomorphic to SEGs. However, as we explain in Section 2, many data-flow problems do not fit into this category; nevertheless, we can handle them sparsely. Secondly, we improve on SEGs in terms of space: this data-structure keeps - for each program variable - a mapping from SEG vertices to Control Flow Graph (CFG) edges that is linear on the size of the CFG. We do not keep this map. Instead, we can replace it with the fast liveness check algorithm that SSA form programs admit [33]. Thus, whenever necessary we can map the information related to a variable to the program points where this variable is live.

We have implemented our framework on top of the LLVM compiler [34], and have used it to provide intermediate representations to two well known compiler optimizations: Wegman *et al.*'s [20] conditional constant propagation, and Bodik *et al.*'s [6] algorithm for array bounds check elimination. We have also built the SSI form as defined by Singer, and compare it with the other program representations that we produce. The intermediate program representations that we derive from our framework increase the size of the original program by less than 5%. This is one order of magnitude less than Singer's SSI form. Furthermore, our experiments indicate that the time to build these program representations is less than 2% of the time taken by the standard suite of optimizations used in the LLVM compiler.

2. Sparse Data-flow Analyses

In this section we quickly review some concepts related to flow analyses. For a more in depth overview of this topic we recommend Nielson *et al.* [35]. The monotone data-flow framework associates *information* with *program points*. We define a *Program Point* as any minimum region in the program code where a data-flow analysis can acquire information. The algorithms that we describe in this paper consider as program points the instructions in the source code, and the regions between consecutive instructions. A *transfer function* determines how information flows between adjacent program points. This information is an element of an algebraic body called a *lattice*. For instance, liveness analysis is a flow problem in which the chal-

lenge is to determine which variables are *live*[1] in and out of each CFG node. The regions of interest, in this case, are *program points between instructions*. A variable v is *live out* of an instruction *inst* if there is a path from *inst* to another instruction *inst'* that uses v, and v is not re-defined along this path. A variable is *live in* at an instruction *inst* if it is live out at *inst*, and it is not defined by *inst*. The result of liveness analysis is a mapping that gives, for each instruction, its IN and OUT sets. We will focus on liveness analysis for a single variable v, e.g., either $IN =$ Live, or $IN =$ Dead; same for OUT. Normally we find a solution to a data-flow problem by continuously solving a set of data-flow equations associated with each program region until a fix point is reached. Given a transfer function f_v^{inst}, and a meet operator $\wedge$ that we will define later, regarding range analysis these equations are:

$$\begin{cases} IN[inst] & = & f_v^{inst}(OUT[inst]) \\ OUT[inst] & = & \bigwedge_{S \in succ(inst)} IN[S] \end{cases} \tag{1}$$

Because liveness analysis combines information that flows out of a node to find the information that flows into it, we call it a *backward* analysis. Forward analyses are the opposite: the meet operator combines the information that comes from the predecessors of a region to produce the information that flows to the successors of this region. Different types of program instructions are associated with different transfer functions. In the case of liveness analysis, we have three types of transfer functions, which depend on the instruction either using or defining the variable v:

Type of instruction *inst*	Transfer function
inst uses v	$f_v^{inst} = \lambda x.\text{Live}$
inst defines v and does not use v	$f_v^{inst} = \lambda x.\text{Dead}$
inst neither uses nor defines v	$f_v^{inst} = \lambda x.x$

Some program points are considered *meet nodes*, because they combine the information that comes from two or more regions. In the case of liveness analysis, conditional branches are meet nodes, because they are source of two different program paths. The variable of interest v may be live in one of

[1]Since "live" is a technical term, we use it, instead of "alive", even as a predicate adjective, e.g., "the variable is live".

these paths, and dead along the other. Information, in this case, is combined via the meet operator $\wedge$. For liveness analysis, this operator is defined by the table below, which says, for instance, that if a variable is dead along a path and live along the other, then it is live past that meet point:

$\wedge$	Dead	Live
Dead	Dead	Live
Live	Live	Live

Some transfer functions are identities. For instance, in liveness analysis, an instruction that neither defines nor uses any variable is associated with an identity transfer function. The goal of sparse data-flow analysis is to shortcut these functions, a task that we accomplish by grouping contiguous program points bound to identities into larger regions. Sometimes it is possible to perform this grouping more efficiently via a customized program representation [11, 21, 22, 36]. In particular, the class of Partitioned Data-flow Analyses (PDA), defined by Zadeck [32], greatly benefits from sparsity. These analyses, which include live variables, reaching definitions and forward/backward printing, can be decomposed into a set of sparse data-flow problems – usually one per variable – each independent on the other. For completeness, we re-state Zadeck's definition, as the sum of two notions: Partitioned Variable Problem (PVP) and Partitioned Variable Lattice (PVL).

Property 1 (PVP/PVL). Partitioned Variable Problem:
Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be the set of program variables. We consider, without loss of generality, a forward data-flow analysis. This data-flow analysis is an equation system that associates, with each program point i, an element of a lattice $\mathcal{L}$, given by the equation $[x]^i = \bigwedge_{s \in pred(i)} F^s([x]^s)$, where $[x]^i$ denotes the abstract state associated with variable x at program point i, and F^s is the transfer function associated with program point s. The analysis can be written[2] as a constraint system that binds to each program point i and each $s \in pred(i)$ the equation $[x]^i = [x]^i \wedge F^s([x]^s)$ or, equivalently, the inequation $[x]^i \sqsubseteq F^s([x]^s)$. The corresponding Maximum Fixed Point (MFP) problem is said to be a *Partitioned Variable Problem* iff:

[2]As far as we are concerned with finding its maximum solution. See for example Section 1.3.2 of [35].

[PVL]: $\mathcal{L}$ can be decomposed into the product of $\mathcal{L}_{v_1} \times \ldots \times \mathcal{L}_{v_n}$ where each $\mathcal{L}_{v_i}$ is the lattice associated with program variable v_i.

[PVP]: each transfer function F^s can also be decomposed into a product $F^s_{v_1} \times F^s_{v_2} \times \ldots \times F^s_{v_n}$ where $F^s_{v_j}$ is a function from $\mathcal{L}_{v_j}$ to $\mathcal{L}_{v_j}$.

Liveness analysis is a partitioned variable problem: the liveness information (lattice of Boolean values $\mathcal{B}$) can be computed for each *individual* (PVL property) variable *independently* (PVP property): the overall lattice can be written as a cross product $\mathcal{L} = \mathcal{B}^n$. The liveness information for variable v at program point i, e.g., $[v]^i$, can be expressed in term of its state at the successors s of i: $[v]^i = [v]^i \wedge F^s_v([v]^s)$ with F^s_v from $\mathcal{B}$ to $\mathcal{B}$.

Many data-flow analyses do *not* provide the PVP property; however, most of them *do* fulfill the PVL property. Consider a problem as simple as constant propagation as an example: if we denote by $\mathcal{C}$ the lattice of constants, the overall lattice can be written as $\mathcal{L} = \mathcal{C}^n$ with n the number of variables (PVL property); as opposed to liveness information, the constant value of some variable v at program point i has to be expressed in term of the constant value of *some other* variables (not only v) at the predecessors S of i: $[v]^i = [v]^i \wedge F^s_v([v_1]^s, \ldots, [v_n]^s)$ with F^s_v from $\mathcal{L}$ to $\mathcal{B}$ (and not from $\mathcal{B}$ to $\mathcal{B}$). Notice that there are data-flow analyses that do not meet the PVL property, such as those that rely on relations between variables [37].

If the information associated with a variable is invariant along its entire live range, then we can bind this information to the variable itself. In other words, we can replace all the constraint variables $[v]^i$ by a single constraint variable $[v]$, for each variable v and every $i \in \text{live}(v)$. In the context of constant propagation, at the program points $s \in \text{live}(v)$ that do not redefine a variable v, $[v]^i = [v]^i \wedge F^s_v([v_1]^s, \ldots, [v_n]^s) = [v]^i \wedge [v]^s$ simplifies into $[v] = [v]$. On the other hand, $F_v^{def(v)}$ simplifies to a function that depends only on some $[u]$ where each u is an argument of the instruction defining v. This gives the intuition on why a propagation engine along the def-use chains of a SSA-form program can be used to solve the constant propagation problem in an equivalent, yet "sparser", manner. This also paves the way toward a formal definition of the Static Single Information property.

Property 2 (SSI). Static Single Information: Consider a forward (resp. backward) monotone PVL problem E_{dense} stated as a set of constraints $[v]^i \sqsubseteq F^s_v([v_1]^s, \ldots, [v_n]^s)$ for every variable v, each program point i,

and each $s \in pred(i)$ (resp. $s \in succ(i)$). A program representation fulfills the Static Single Information property iff:

[SPLIT]: for each variable v, each $s \in \text{live}(v)$ such that $F_v^s \neq \lambda x.\bot$ is non-trivial, i.e. is not the simple projection on $\mathcal{L}_v$ (see Definition 2 in Appendix B), should contain a definition (resp. last use) of v; Let $(Y_v^i)_{(v,i) \in variables \times prog\_points}$ be a maximum solution to E_{dense}. Each join (resp. split) node for which $F_v^s(Y_{v_1}^s, \ldots, Y_{v_n}^s)$ has different values on its incoming edges should have a ϕ-function (resp. σ function) for v as defined in Section 2.1.

[INFO]: each program point $i \notin \text{live}(v)$ should be bound to an undefined (see Definition 2) transfer function, e.g., $F_v^s = \lambda x.\bot$.

[LINK]: each instruction *inst* for which F_v^{inst} depends on some $[u]^s$ (see Definition 2) should contain a (potentially pseudo) use (resp. def) of u live-out (resp. live-in) of *inst*.

[VERSION]: for each variable v, live(v) is a connected component of the CFG.

These properties allows us to attach the information to variables, instead of program points. The SPLIT property forces the information related to a variable to be invariant along its entire live-range. INFO forces this information to be irrelevant outside the live range of the variable. The LINK property forces the def-use chains to reach the points where information is available for a transfer function to be evaluated. The VERSION property provides an one-to-one mapping between variable names and live ranges.

We must split live ranges to provide the SSI properties. If we split them between each pair of consecutive instructions, then we would automatically provide these properties, as the newly created variables would be live at only one program point. However, this strategy would lead to the creation of many trivial program regions, and we would lose sparsity. In Section 3 we provide a sparser way to split live ranges that fit Property 2. Possibly, we may have to extend the live-range of a variable to cover every program point where the information is relevant. We accomplish this last task by inserting into the program pseudo-uses and pseudo-definitions of this variable.

2.1. Special instructions used to split live ranges

We group program points in three kinds: interior nodes, branches and joins. At each place we use a different notation to denote live range splitting.

Interior nodes are program points that have a unique predecessor and a unique successor. At these points we perform live range splitting via copies. If the program point already contains another instruction, then this copy *must* be done *in parallel* with the existing instruction. The notation,

$$inst \;\parallel\; v_1 = v_1' \;\parallel\; \ldots \;\parallel\; v_m = v_m'$$

denotes m copies $v_i = v_i'$ performed in parallel with instruction $inst$. This means that all the uses of $inst$ plus all v_i' are read simultaneously, then $inst$ is computed, then all definitions of $inst$ plus all v_i are written simultaneously.

We call *joins* the program points that have one successor and multiple predecessors. For instance, two different definitions of the same variable v might be associated with two different constants; hence, providing two different pieces of information about v. To avoid that these definitions reach the same use of v we merge them at the earliest program point where they meet. We do it via special instructions called ϕ-functions, which were introduced by Cytron *et al.* to build SSA-form programs [26]. The assignment

$$v_1 = \phi(v_1^1 : l^1, \ldots, v_1^q : l^q) \;\parallel\; \ldots \;\parallel\; v_m = \phi(v_m^1 : l^1, \ldots, v_m^q : l^q)$$

contains m ϕ-functions to be performed in parallel. The ϕ symbol works as a multiplexer. It will assign to each v_i the value in v_i^j, where j is determined by l^j, the basic block last visited before reaching the ϕ assignment. The above statement encapsulates m parallel copies: all the variables $v_1^j, \ldots, v_m^j$ are simultaneously copied into the variables $v_1, \ldots, v_m$.

In backward analyses the information that emerges from different uses of a variable may reach the same *branch point*, which is a program point with a unique predecessor and multiple successors. To ensure Property 2, the use that reaches the definition of a variable must be unique, in the same way that in a SSA-form program the definition that reaches a use is unique. We ensure this property via special instructions that Ananian has called σ-functions [27]. The σ-functions are the dual of ϕ-functions, performing a parallel assignment depending on the execution path taken. The assignment

$$(v_1^1 : l^1, \ldots, v_1^q : l^q) = \sigma(v_1) \;\parallel\; \ldots \;\parallel\; (v_m^1 : l^1, \ldots, v_m^q : l^q) = \sigma(v_m)$$

represents m σ-functions that assign to each variable v_i^j the value in v_i if control flows into block l^j. These assignments happen in parallel, i.e., the m σ-functions encapsulate m parallel copies. Also, notice that variables live in different branch targets are given different names by the σ-function that ends that basic block.

2.2. Propagation engine

As mentioned earlier for any program that fulfills the SSI property of a given PVL problem, a propagation engine along the def-use chains can be used to solve it sparsely. Let us consider a unidirectional forward (resp. backward) PVL problem E^{ssi}_{dense} stated as a set of equations $[v]^i \sqsubseteq F^s_v([v_1]^s, \ldots, [v_n]^s)$ (or equivalently $[v]^i = [v]^i \wedge F^s_v([v_1]^s, \ldots, [v_n]^s)$ for every variable v, each program point i, and each $s \in pred(i)$ (resp. $s \in succ(i)$).

We see two ways to handle ϕ and σ-functions during the data-flow analysis. Either we consider each of them as a whole and get only one equation per ϕ or σ-function, or we consider them as a set of copies and then have as many equations as the number of parameters, in the case of ϕ-functions, or successors, in the case of σ-functions. We have opted for the second choice, because it simplifies our notation. Any ϕ-function $a = \phi(a_1 : l^1, \ldots, a_m : l^m)$ (resp. σ-function $(a_1 : l^1, \ldots, a_m : l^m) = \sigma(a)$) at program point i leads to as many constraints as the set of predecessors (resp. successors) S_j of i. In other words, a ϕ-function such as $a = \phi(a_1 : l^1, \ldots, a_m : l^m)$, gives us n constraints such as $[a]^i \sqsubseteq [a_j]^{l_j}$, which we can simplify into the classical meet $[a]^i \sqsubseteq \bigwedge_{l_j \in pred(i)} [a_j]^{l_j}$. Similarly, a σ-function $(S_1 : a_1, \ldots, S_m : a_m) = \sigma(a)$ at program point i yields n constraints such as $[a_j]^{l_j} \sqsubseteq [a]^i$.

Given a program that fulfills the SSI property for E^{ssi}_{dense} and the set of transfer functions F^s_v, we show here how to build an equivalent sparse constrained system.

Definition 1 (SSI constrained system). Consider that a program in SSI form gives us a constraint system that associates with each variable v the constraints $[v]^i = [v]^i \wedge F^s_v([v_1]^s, \ldots, [v_n]^s)$. We define a system of sparse equations E^{ssi}_{sparse} as follows:

- For each instruction at a program point i that defines (resp. uses) a variable v, we let $a \ldots b$ be its set of used (resp. defined) variables. Because of the LINK property, F^s_v depends only on some $[a]^s \ldots [b]^s$. Thus, there exists a function G^s_v defined as the restriction of F^s_v on $\mathcal{L}_a \times \ldots \times \mathcal{L}_b$, i.e. $G^s_v([a], \ldots, [b]) = F^s_v([v_1], \ldots, [v_n])$.

- The sparse constrained system associates with each variable v, and each definition (resp. use) point s of v, the corresponding constraint $[v] \sqsubseteq G_v^s([a], \ldots, [b])$ where $a, \ldots, b$ are used (resp. defined) at s.

The SSI constrained system might have several inequations for the same left-hand-side. This is due to the way that we handle ϕ and σ functions. Our definition of the SSI property, as opposed to the original ones [27, 28], does not ensure the SSA or the SSU properties, because such a guarantee is not necessary to every sparse analysis. It is a common assumption in the compiler's literature that "data-flow analysis (...) can be made simpler when each variable has only one definition", as stated in Chapter 19 of Appel's textbook [38]. A naive interpretation of the above statement could lead one to conclude that data-flow analyses become simpler as soon as the program representation enforces a single source of information per live-range: SSA for forward propagation, SSU for backward, and the *original* SSI bi-directional analyses. This premature conclusion is contradicted by the example of dead-code elimination, a backward data-flow analysis that the SSA form simplifies. In fact, the SSA form fulfills our definition of the SSI property for dead-code elimination. Nevertheless, the corresponding constraint system has several inequations (one per variable use) for the same left-hand-side (one for each variable). It is well known that such a system can be solved using chaotic iteration such as the worklist algorithm [35, Sec 6.1] given in Figures 1 and 2: replace G_v^i in Figure 1 by "*i is a useful instruction or one of its definitions is marked as useful*" and one obtains the classical algorithm for dead-code elimination.

The following theorem proved in Appendix C states the equivalence between sparse and dense analyses.

Theorem 1 (sparse ≡ dense). *Consider a program in SSI-form that gives origin to a constraint system E_{dense}^{ssi} associating with each variable v the constraints $[v]^i = [v]^i \wedge F_v^s([v_1]^s, \ldots, [v_n]^s)$. Suppose that each F_v^s is a monotone function from $\mathcal{L}^n$ to $\mathcal{L}$ where $\mathcal{L}$ is of finite height. Let $(Y_v)_{v \in variables}$ be the maximum solution of the corresponding sparse constraint system.*

Then, $(X_v^i)_{(v,i) \in variables \times prog\_points}$ with $\begin{cases} X_v^i = Y_v & \text{for } i \in live(v) \\ X_v^i = \bot & \text{otherwise} \end{cases}$ is the maximum solution to E_{dense}^{ssi}.

2.3. Examples of sparse data-flow analyses

As we have mentioned before, many data-flow analyses can be classified as PVP/PVL problems. In this section we present some meaningful examples.

```
1  function back_propagate(transfer_functions 𝒢)
2      worklist = ∅
3      foreach v ∈ vars: [v] = ⊤
4      foreach i ∈ insts: worklist += i
5      while worklist ≠ ∅:
6          let i ∈ worklist; worklist −= i
7          foreach v ∈ i.uses():
8              [v]_new = [v] ∧ G^i_v([i.defs()])
9              if [v] ≠ [v]_new:
10                 stack += v.defs()
11                 [v] = [v]_new
```

Figure 1: Backward propagation engine under SSI

```
1  function forward_propagate(transfer_functions 𝒢)
2      worklist = ∅
3      foreach v ∈ vars: [v] = ⊤
4      foreach i ∈ insts: worklist += i
5      while worklist ≠ ∅:
6          let i ∈ worklist; worklist −= i
7          foreach v ∈ i.defs():
8              [v]_new = [v] ∧ G^i_v([i.uses()])
9              if [v] ≠ [v]_new:
10                 stack += v.uses()
11                 [v] = [v]_new
```

Figure 2: Forward propagation engine under SSI

Class Inference. Some dynamically typed languages, such as Python, JavaScrip, Ruby or Lua, represent objects as tables containing methods and fields. It is possible to improve the execution of programs written in these languages if we can replace these simple tables by actual classes with virtual tables [39]. A class inference engine tries to assign a class to a variable v based on the ways that v is used. The Python program in Figure 3(a) illustrates this optimization. Our objective is to infer the correct suite of methods for each object bound to variable v. Figure 3(b) shows the control flow graph of the program, and Figure 3(c) shows the results of a dense implementation of this analysis. Notice that each program instruction is associated with a transfer function, and that some of these functions, such as that in label l_3,

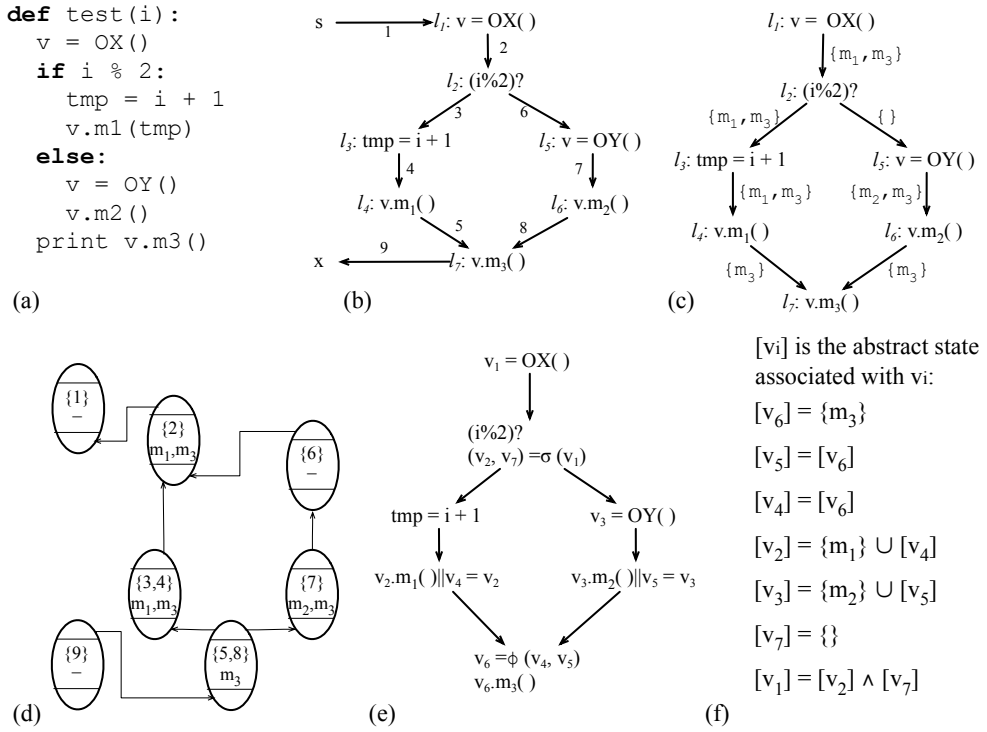

Figure 3: Class inference analysis as an example of backward data-flow analysis that takes information from the uses of variables.

are trivial, having no influence on the data-set that they create. Choi *et al.* [21] would perform the class inference analysis on the SEG in Figure 3(d). Each node of this graph is labeled with the edges of the CFG that it groups together. All the CFG edges grouped by a node have the same data-flow information, as one can verify in Figure 3(c). We show this information – a set of methods – in each SEG node. The SEG edges point in the direction that information flows between program regions. Instead of using a separate data-structure, like Choi *et al.* do, we work directly on the program, producing the representation given in Figure 3(e). Because type inference is a backward analysis that extracts information from use sites, we split live ranges at these program points, and rely on σ-functions to merge them back. We want to stay in SSA-form; hence, we must also insert ϕ-function to join the live ranges that denote the same variable definition. The use-def chains that we derive from the program representation lead naturally to a constraint system, which we

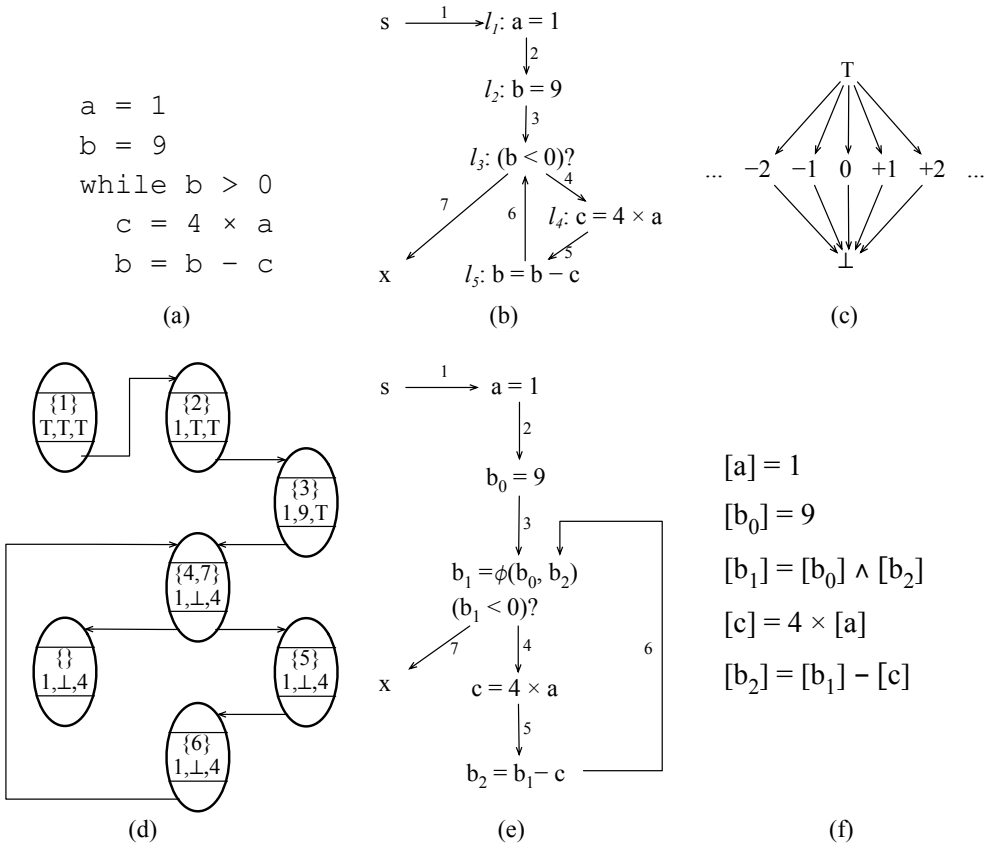

Figure 4: Constant propagation as an example of forward data-flow analysis that takes information from the definitions of variables.

show in Figure 3(f), where $[v_j]$ is the information associated with variable v_j. A fix-point to this constraint system is a solution to our data-flow problem. Class inference is a Partitioned Variable Problem (PVP)[3], because the data-flow information associated with a variable v can be computed independently from the other variables. In the words of Choi *et al.*, SEGs are "specially attractive" for this kind of problem.

[3]Actually class inference is no more a PVP as soon as we want to propagate the information through copies.

Constant Propagation. There exist many data-flow analyses that are not Partitioned Variable Problems. Constant propagation is an example: in this analysis, the abstract state of a variable v is determined by the abstract state of the variables used to define v. Figure 4 illustrates constant propagation. We want to find out which variables in the program of Figure 4(a) can be replaced by constants. The CFG of this program is given in Figure 4(b). Constant propagation has a very simple lattice, which we show in Figure 4(c). The SEG created for this instance of constant propagation is given in Figure 4(d). Every instruction in this example either generates information, or merges it; thus, the SEG contains a node representing each instruction. We have augmented each SEG node with the edges that it represent in the CFG, plus the final result of the constant propagation problem in that region. Because we have three variables, each node is associated with a three dimensional vector $([a], [b], [c])$, where $[x]$ is the abstract state of variable x, as given by the lattice in Figure 4(c). Our approach, to this kind of problem is sparser, because we bind a lattice value directly to each live range, instead of having to associate product lattices to program regions. In constant propagation, information is produced at the program points where variables are defined. Thus, in order to provide Property 2, we must guarantee that each program point is dominated by a single definition of a variable. Figure 4(e) shows the intermediate representation that we create for the program in Figure 4(b). In this case, our intermediate representation is equivalent to the SSA form. The def-use chains implicit in our program representation lead to the constraint system shown in Figure 4(f).

Taint analysis. The objective of taint analysis [15] is to find program vulnerabilities. In this case, a harmful attack is possible when input data reaches sensitive program sites without going through special functions called sanitizers. Figure 5 illustrates this type of analysis. We have used ϕ and σ-functions to split the live ranges of the variables in Figure 5(a) producing the program in Figure 5(b). Lets assume that *echo* is a sensitive function, because it is used to generate web pages. For instance, if the data passed to *echo* is a JavaScript program, then we could have an instance of cross-site scripting attack. Thus, the statement *echo* v_1 may be a source of vulnerabilities, as it outputs data that comes directly from the program input. On the other hand, we know that *echo* v_2 is always safe, for variable v_2 is initialized with a constant value. The call *echo* v_5 is always safe, because variable v_5 has been sanitized; however, the call *echo* v_4 might be tainted, as variable v_4 results

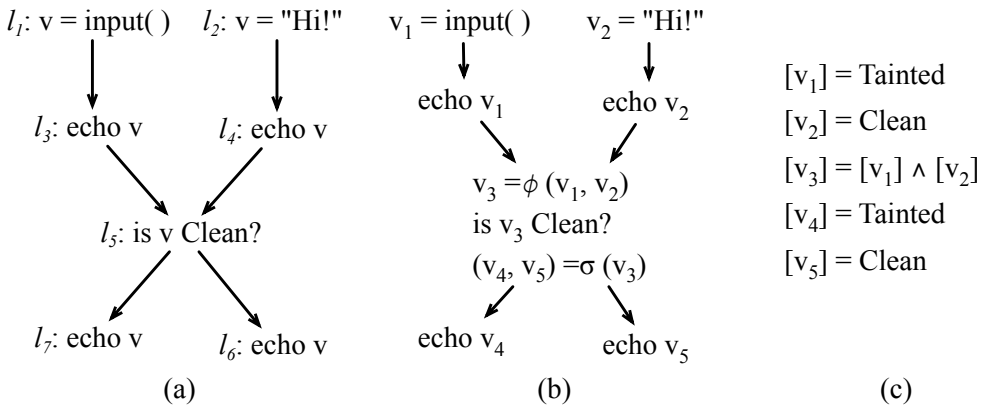

Figure 5: Taint analysis as an example of forward data-flow analysis that takes information from the definitions of variables and conditional tests on these variables.

from a failed attempt to sanitize v. The def-use chains that we derive from the program representation leads naturally to a constraint system, which we show in Figure 5(c). The intermediate representation that we create in this case is equivalent to the *Extended Single Static Assignment* (e-SSA) form [6]. It also suits the ABCD algorithm for array bounds-checking elimination [6], Su and Wagner's range analysis [19] and Gawlitza *et al.*'s range analysis [40].

Null pointer analysis. The objective of null pointer analysis is to determine which references may hold null values. Nanda and Sinha have used a variant of this analysis to find which method dereferences may throw exceptions, and which may not [13]. This analysis allows compilers to remove redundant null-exception tests and helps developers to find null pointer dereferences. Figure 6 illustrates this analysis. Because information is produced at use sites, we split live ranges after each variable is used, as we show in Figure 6(b). For instance, we know that the call $v_2.m()$ cannot result in a null pointer dereference exception, otherwise an exception would have been thrown during the invocation $v_1.m()$. On the other hand, in Figure 6(c) we notice that the state of v_4 is the meet of the state of v_3, definitely not-null, and the state of v_1, possibly null, and we must conservatively assume that v_4 may be null.

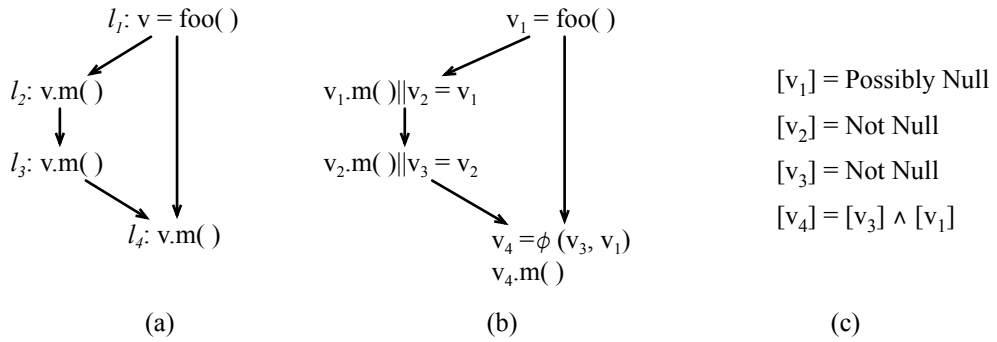

Figure 6: Null pointer analysis as an example of forward data-flow analysis that takes information from the definitions and uses of variables.

3. Building the Intermediate Program Representation

A *live range splitting strategy* $\mathcal{P}_v = I_\uparrow \cup I_\downarrow$ over a variable v consists of a set of "oriented" program points. We let $I_\downarrow$ denote a set of points i with forward direction. Similarly, we let $I_\uparrow$ denote a set of points i with backward direction. The live-range of v must be split at least at every point in $\mathcal{P}_v$. Going back to the examples from Section 2.3, we have the live range splitting strategies enumerated below. The list in Figure 7 gives further examples of live range splitting strategies.

- Class inference is a backward analysis that takes information from the uses of variables; thus, for each variable, the live-range splitting strategy is characterized by the set $Uses_\uparrow$ where *Uses* is the set of use points. For instance, in Figure 3(e), we have that $\mathcal{P}_v = \{l_4, l_6, l_7\}_\uparrow$.

- Constant propagation is a forward analysis that takes information from definition sites. Thus, for each variable v the live-range splitting strategy is characterized by the set $Defs_\downarrow$ where *Defs* is the set of definition points. For instance, in Figure 4, we have that $\mathcal{P}_b = \{l_2, l_5\}_\downarrow$.

- Taint analysis is a forward analysis that takes information from points where variables are defined, and conditional tests that use these variables. For instance, in Figure 5, we have that $\mathcal{P}_v = \{l_1, l_2, \mathrm{Out}(l_5)\}_\downarrow$ where $\mathrm{Out}(l_i)$ denotes the exit of l_i.

Client	**Splitting strategy $\mathcal{P}$**
Alias analysis, reaching definitions cond. constant propagation [20]	$Defs_{\downarrow}$
Partial Redundancy Elimination [27, 28]	$Defs_{\downarrow} \bigcup LastUses_{\uparrow}$
ABCD [6], taint analysis [15], range analysis [19, 40]	$Defs_{\downarrow} \bigcup \text{Out}(Conds)_{\downarrow}$
Stephenson's bitwidth analysis [18]	$Defs_{\downarrow} \bigcup \text{Out}(Conds)_{\downarrow} \bigcup Uses_{\uparrow}$
Mahlke's bitwidth analysis [12]	$Defs_{\downarrow} \bigcup Uses_{\uparrow}$
An's type inference [41], Class inference [39]	$Uses_{\uparrow}$
Hochstadt's type inference [10]	$Uses_{\uparrow} \bigcup \text{Out}(Conds)_{\uparrow}$
Null-pointer analysis [13]	$Defs_{\downarrow} \bigcup Uses_{\downarrow}$

Figure 7: Live range splitting strategies for different data-flow analyses. We use *Defs* (*Uses*) to denote the set of instructions that define (use) the variable; *Conds* to denote the set of instructions that apply a conditional test on a variable; Out(*Conds*) the exits of the corresponding basic blocks; *LastUses* to denote the set of instructions where a variable is used, and after which it is no longer live.

- Nanda *et al.*'s null pointer check [13] is a forward analysis that takes information from definitions and uses. For instance, in Figure 6, we have that $\mathcal{P}_v = \{l_1, l_2, l_3, l_4\}_{\downarrow}$.

The algorithm SSIfy in Figure 8 implements a live range splitting strategy in three steps. Firstly, it splits live ranges, inserting new definitions of variables into the program code. Secondly, it renames these newly created definitions; hence, ensuring that the live ranges of two different re-definitions of the same variable do not overlap. Finally, it removes dead and non-initialized definitions from the program code. We describe each of these phases in the rest of this section.

Splitting live ranges through the creation of new definitions of variables. In order to implement $\mathcal{P}_v$ we must split the live ranges of v at each program point listed by $\mathcal{P}_v$. However, these points are not the only ones where splitting might be necessary. As we have pointed out in Section 2.1, we might have, for the same original variable, many different sources of information reaching

```
1  function SSIfy(var v, Splitting_Strategy P_v)
2      split(v, P_v)
3      rename(v)
4      clean(v)
```

Figure 8: Split the live ranges of v to convert it to SSI form

a common program point. For instance, in Figure 4(b), there exist two definitions of variable b – l_2 and l_5 – that reach the use of b at l_3. The information that flows forward from l_2 and l_5 collides at l_3, the merge point of the if-then-else. Hence the live-range of b has to be split immediately before l_3 – at $\text{In}(l_3)$ –, leading, in our example, to a new definition b_1. In general, the set of program points where information collides can be easily characterized by join sets [42]. The join set of a set of nodes P contains the CFG nodes that can be reached by two or more nodes of P through disjoint paths. Join sets created by the forward propagation of information can be over-approximated via the notion of iterated dominance frontier [43]. This concept is the basics of SSA construction, and for completeness we recall its definition below:

- **Dominance**: a CFG node n dominates a node n' if every program path from the entry node of the CFG to n' goes across n. If $n \neq n'$, then we say that n *strictly* dominates n'.

- **Dominance frontier** (DF): a node n' is in the dominance frontier of a node n if n dominates a predecessor of n', but does not strictly dominate n'.

- **Iterated dominance frontier** (DF^+): the iterated dominance frontier of a node n is the limit of the sequence:

$$\begin{array}{rcl} DF_1 & = & DF(n) \\ DF_{i+1} & = & DF_i \cup \{DF(z) \mid z \in DF_i\} \end{array}$$

Similarly, split sets created by the backward propagation of information can be over-approximated by the notion of *iterated post-dominance frontier* (pDF^+), which is the dual of DF^+ [38]. That is, the post-dominance frontier is the dominance frontier in a CFG where direction of edges have been reversed.

```
1  function split(var v, Splitting_Strategy P_v = I_↓ ∪ I_↑)
2        "compute the set of split points"
3        S_↑ = ∅
4        foreach i ∈ I_↑:
5              if i.is_join:
6                    foreach e ∈ incoming_edges(i):
7                          S_↑ = S_↑ ⋃ Out(pDF+(e))
8              else:
9                    S_↑ = S_↑ ⋃ Out(pDF+(i))
10       S_↓ = ∅
11       foreach i ∈ S_↑ ⋃ Defs(v) ⋃ I_↓:
12             if i.is_branch:
13                   foreach e ∈ outgoing_edges(i)
14                         S_↓ = S_↓ ⋃ In(DF+(e))
15             else:
16                   S_↓ = S_↓ ⋃ In(DF+(i))
17       S = P_v ⋃ S_↑ ⋃ S_↓
18       "Split live range of v by inserting φ, σ, and copies"
19       foreach i ∈ S:
20             if i does not already contain any definition of v:
21                   if i.is_join: insert "v = φ(v, ..., v)" at i
22                   elseif i.is_branch: insert "(v, ..., v) = σ(v)" at i
23                   else: insert a copy "v = v" at i
```

Figure 9: Live range splitting. We use $\text{In}(l)$ to denote a program point immediately before l, and $\text{Out}(l)$ to denote a program point immediately after l.

Figure 9 shows the algorithm that we use to create new definitions of variables. This algorithm has three main phases. First, in lines 3-9 we create new definitions to split the live ranges of variables due to backward collisions of information. These new definitions are created at the iterated post-dominance frontier of points that originate information. If a program point is a join node, then each of its predecessors will contain the live range of a different definition of v, as we ensure in line 6 of our algorithm. Notice that these new definitions are not placed parallel to an instruction, but in the region immediately after it, which we denote by $\text{Out}(\ldots)$. In lines 10-16 we perform the inverse operation: we create new definitions of variables due to the forward collision of information. Our starting points, in this case, include also the original definitions of v, as we see in line 11, because we want to stay

in SSA form in order to have access to a fast liveness check [33]. Finally, in lines 17-23 we actually insert the new definitions of v. These new definitions might be created by σ functions (due exclusively to the splitting in lines 3-9); by ϕ-functions (due exclusively to the splitting in lines 10-16); or by parallel copies. Contrary to Singer's algorithm, originally designed to produce SSI form programs, we do not iterate between the insertion of ϕ and σ functions. Nevertheless, as we show in the Appendix, our method is enough to ensure the SSI properties for any combination of unidirectional problems.

Variable Renaming. The algorithm in Figure 10 builds def-use and use-def chains for a program after live range splitting. This algorithm is similar to the standard algorithm used to rename variables during the SSA construction [38, Algorithm 19.7]. To rename a variable v we traverse the program's dominance tree, from top to bottom, stacking each new definition of v that we find. The definition currently on the top of the stack is used to replace all the uses of v that we find during the traversal. If the stack is empty, this means that the variable is not defined at this point. The renaming process replaces the uses of undefined variables by $\perp$ (line 3). We have two methods, stack.set_use and stack.set_def to build the chain relations between the variables. Notice that sometimes we must rename a single use inside a ϕ-function, as in lines 19-20 of the algorithm. For simplicity we consider this single use as a simple assignment when calling stack.set_use, as one can see in line 20. Similarly, if we must rename a single definition inside a σ-function, then we treat it as a simple assignment, like we do in lines 15-16 of the algorithm.

Dead and Undefined Code Elimination. The algorithm in Figure 11 eliminates ϕ-functions that define variables not actually used in the code, σ-functions that use variables not actually defined in the code, and parallel copies that either define or use variables that do not reach any actual instruction. We mean by "actual" instructions, those instructions that already existed in the program before we transformed it with split. In line 3 we let "web" be the set of versions of v, so as to restrict the cleaning process to variable v, as we see in lines 4-6 and lines 10-12. The set "active" is initialized to actual instructions in line 4. Then, during the loop in lines 5-8 we add to active ϕ-functions, σ-functions, and copies that can reach actual definitions through use-def chains. The corresponding version of v is then marked as *defined* (line 8). The next loop, in lines 11-14 performs a similar process, this time to add to the active set, instructions that can reach actual

```
1  function rename(var v)
2        "Compute use-def & def-use chains"
3        "We consider here that stack.peek() = ⊥ if stack.isempty(),
4          and that def(⊥) = entry"
5        stack = ∅
6        foreach CFG node n in dominance order:
7              if exists v = φ(v : l^1, ..., v : l^q) in In(n):
8                    stack.set_def(v = φ(v : l^1, ..., v : l^q))
9              foreach instruction u in n that uses v:
10                   stack.set_use(u)
11             if exists instruction d in n that defines v:
12                   stack.set_def(d)
13             foreach instruction (...) = σ(v) in Out(n):
14                   stack.set_use((...) = σ(v))
15             if exists (v : l^1, ..., v : l^q) = σ(v) in Out(n):
16                   foreach v : l^i = v in (v : l^1, ..., v : l^q) = σ(v):
17                         stack.set_def(v : l^i = v)
18             foreach m in successors(n):
19                   if exits v = φ(..., v : l^n, ...) in In(m):
20                         stack.set_use(v = v : l^n)
21 function stack.set_use(instruction inst):
22       while def(stack.peek()) does not dominate inst: stack.pop()
23       v_i = stack.peek()
24       replace the uses of v by v_i in inst
25       if v_i ≠ ⊥: set Uses(v_i) = Uses(v_i) ⋃ inst
26 function stack.set_def(instruction inst):
27       let v_i be a fresh version of v
28       replace the defs of v by v_i in inst
29       set Def(v_i) = inst
30       stack.push(v_i)
```

Figure 10: Versioning

uses through def-use chains. The corresponding version of v is then marked as *used* (line 14). Each non live variable (see line 15), i.e. either undefined or dead (non used) is replaced by $\perp$ in all ϕ, σ, or copy functions where it appears in. This is done by lines 15-18. Finally every useless ϕ, σ, or copy functions are removed by lines 19-20. As a historical curiosity, Cytron *et al.*'s procedure to build SSA form produced what is called *the minimal*

```
1  clean(var v)
2        let web = {v_i | v_i is a version of v}
3        let defined = ∅
4        let active = { inst | inst actual instruction and web ∩ inst.defs ≠ ∅}
5        while ∃inst ∈ active s.t. web ∩ inst.defs\defined ≠ ∅:
6              foreach v_i ∈ web ∩ inst.defs\defined:
7                    active = active ∪ Uses(v_i)
8                    defined = defined ∪ {v_i}
9        let used = ∅
10       let active = { inst | inst actual instruction and web ∩ inst.uses ≠ ∅}
11       while ∃inst ∈ active s.t. inst.uses\used ≠ ∅:
12             foreach v_i ∈ web ∩ inst.uses\used:
13                   active = active ∪ Def(v_i)
14                   used = used ∪ {v_i}
15       let live = defined ∩ used
16       foreach non actual inst ∈ Def(web):
17             foreach v_i operand of inst s.t. v_i ∉ live:
18                         replace v_i by ⊥
19             if inst.defs = {⊥} or inst.uses = {⊥}
20                   remove inst
```

Figure 11: Dead and undefined code elimination. Original instructions not inserted by split are called *actual* instruction. We let *inst*.defs denote the set of variable(s) defined by *inst*, and *inst*.uses denote the set of variables used by *inst*.

representation [42]. Some of the ϕ-functions in the minimal representation define variables that are never used. Briggs *et al.* [44] remove these variables; hence, producing what compiler writers normally call *pruned SSA-form*.

3.1. Implementing parallel copies, ϕ and σ-functions

Traditional instruction sets, such as x86 or PowerPC, do not provide ϕ-functions nor σ-functions. Thus, before producing an executable program, the compiler must implement these instructions somehow. Normally, ϕ-functions and parallel copies are replaced by ordinary copy instructions, as discussed by Briggs *et al.* [44] or Benoit *et al* [29]. There exists ways to implement the semantics of parallel copies via simple copies without increasing the register pressure in the source program [45]. The implementation of σ-functions; however, has not been discussed in the literature. A possible solution is to get rid of copies and σ-functions by simply copy-propagating

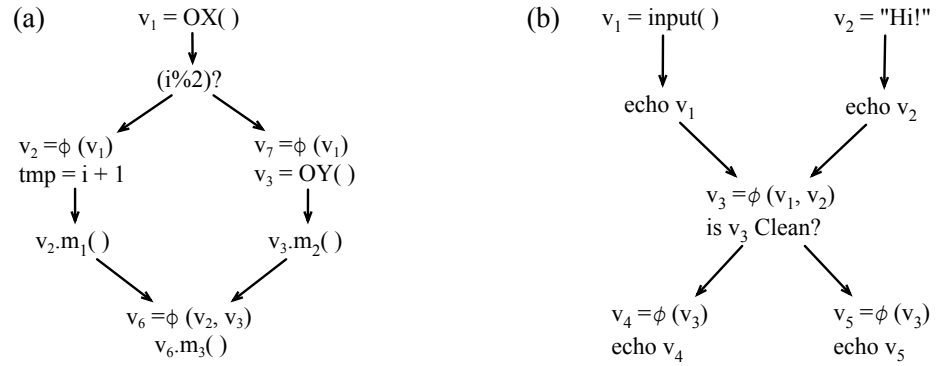

Figure 12: (a) getting rid of copies and σ-functions; (b) implementing σ-functions via single arity ϕ-functions.

them; thus, leaving for the SSA elimination module the task of replacing σ-functions with special instructions. As an example, Figure 12(a) shows the result of copy folding applied on Figure 3(e).

Alternatively, σ-functions can be implemented as single arity ϕ-functions. As an example, Figure 12(b) shows how we would represent the σ-functions in Figure 5(b). If l is a branch point with n successors that would contain a σ-function $(v_1 : l^1, \ldots, v_n : l^n) = \sigma(v)$, then, for each successor l^j of l, we insert at the beginning of l^j an instruction $v_j = \phi(v : l)$. Notice that it is possible that l^j already contains a ϕ-function for v. This case happens when the control flow edge $l \rightarrow l^j$ is *critical*. A critical edge links a basic block with several successors to a basic block with several predecessors. If l^j already contains a ϕ-function $v' = \phi(\ldots, v_j, \ldots)$, then we rename v_j to v.

3.2. Deriving dense information from sparse analyses

We can use our sparse data-flow analysis framework to solve even some data-flow problems that demand information at every program point, such as bitwidth analysis [12, 18, 40, 46]. There exist clients of bit-width analyses that need to know the bit sizes of the variables at particular program points. For instance, Barik *et al.* [47] have designed a bit-width aware register allocator. In this setting, the register pressure at a program point p is the sum of the bit sizes of all the variables live at p. We can support the register allocator of Barik *et al.* by coupling the result of the sparse bit-width analysis with live range information. We can perform this coupling efficiently,

because algorithm SSIfy preserves the single static assignment property.

Preserving the SSA properties is key due to two reasons. First, liveness analysis has a non-iterative implementation for SSA-form programs linear on the program size [38, p.429]. Second, if we only need liveness information for some specific variables, at some specific program points, then there is a fast liveness check for SSA-form programs. The problem of answering the question "is variable v live at program point p" has an algorithm that is $O(U)$, where U is the number of times that v is used in the program code [33]. Over 95% of variables found in common benchmarks are used less than 5 times [33, p.42]; thus, this asymptotic complexity is constant in practice.

4. Experimental Results

This section describes experiments that we have performed to probe the size and the runtime efficiency of the algorithms that we use to build intermediate representations. Our experiments were conducted on a dual core `Intel Pentium D` of 2.80GHz of clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [34], and it passes all the tests that LLVM does. The LLVM test suite consists of over 1.3 million lines of C code. In this paper we will be showing only the results of compiling SPEC CPU 2000. In order to compare different live range splitting strategies we generate program representations to three different LLVM analyses:

1. *SSI*: We use Ananian's Static Single Information form [27] as a baseline for our experiments. We build the SSI program representation via Singer's iterative algorithm.
2. *ABCD*: $(\{def, cond\}_{\downarrow})$. This live range splitting strategy generalizes the ABCD algorithm for array bounds checking elimination [6]. An example of this live range splitting strategy is given in Figure 5.
3. *CCP*: $(\{def, cond_{eq}\}_{\downarrow})$. This live range splitting strategy, which supports Wegman *et al.*'s [20] conditional constant propagation, is a subset of the previous strategy. Differently of the ABCD client, this client requires that only variables used in equality tests, e.g., `==`, undergo live range splitting. That is, $cond_{eq}(v)$ denotes the conditional tests that check if v equals a given value.

For an explanation about the sets *defs, uses* and *conds*, see Figure 7.

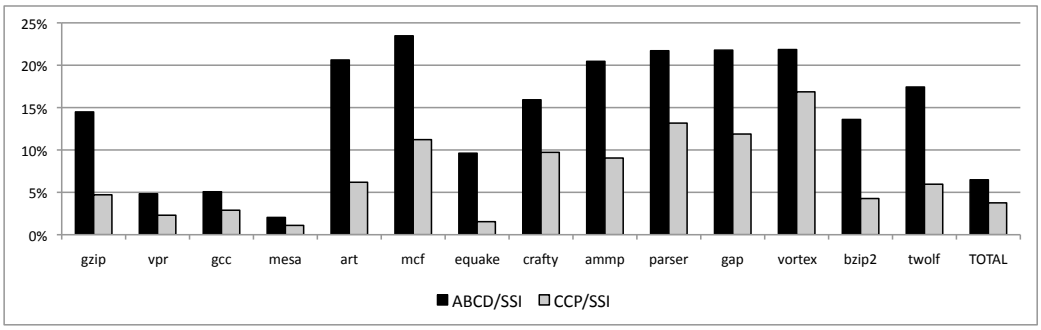

Figure 13: Comparison of the time taken to produce the different program representations. 100% is the time of using the SSI live range splitting strategy. The shorter the bar, the faster the live range splitting strategy. The SSI conversion took 1315.2s in total, the ABCD conversion took 85.2s, and the CCP conversion took 49.4s.

4.1. Runtime

The chart in Figure 13 compares the execution time of the three live range splitting strategies. We show only the time to perform live range splitting. The time to execute the optimization itself, removing array bounds check or performing constant propagation, is not shown. The bars are normalized to the running time of the *SSI* live range splitting strategy. On the average, the ABCD client runs in 6.8% and the CCP client runs in 4.1% of the time of SSI. These two forward analyses tend to run faster in benchmarks with sparse control flow graphs, which present fewer conditional branches, and therefore fewer opportunities to restrict the ranges of variables.

In order to put the time reported in Figure 13 in perspective, Figure 14 compares the running time of our live range splitting algorithms with the time to run the other standard optimizations in our baseline compiler[4]. In our setting, LLVM -O1 runs 67 passes, among analysis and optimizations, which include partial redundancy elimination, constant propagation, dead code elimination, global value numbering and invariant code motion. We believe that this list of passes is a meaningful representative of the optimizations that are likely to be found in an industrial strength compiler. The bars are normalized to the optimizer's time, which consists of the time taken by

[4]To check the list of LLVM's target independent optimizations try `llvm-as < /dev/null | opt -std-compile-opts -disable-output -debug-pass=Arguments`

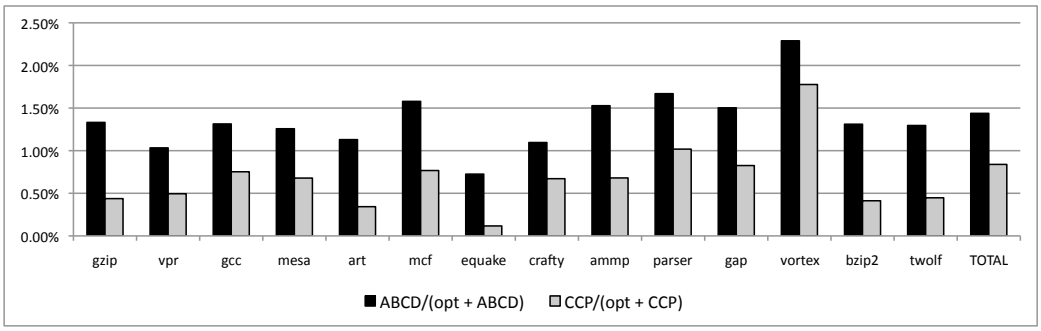

Figure 14: Execution time of two different live range splitting strategies compared to the total time taken by machine independent LLVM optimization passes (`opt`). 100% is the total time taken by `opt`. The shorter the bar, the faster the conversion.

machine independent optimizations plus the time taken by one of the live range splitting clients, e.g, ABCD or CCP. The ABCD client takes 1.48% of the optimizer's time, and the CCP client takes 0.9%. To emphasize the speed of these passes, we notice that the bars do not include the time to do machine dependent optimizations such as register allocation.

4.2. Space

Figure 15 compares the number of ϕ and σ-functions inserted by each live range splitting strategy. The bars are the sum of these instructions, as inserted by each conversion, divided by the number of σ and ϕ-functions inserted by the SSI live range splitting strategy. The CCP client created 67.3K σ-functions, and 28.4K ϕ-functions. The ABCD client created 98.8K σ-functions, and 42.0K ϕ-functions. The SSI conversion inserted 697.6K σ-functions, and 220.6K σ-functions.

The chart in Figure 16 shows the number of σ and ϕ-functions that each live range splitting strategy inserts per variable. The denominator includes only variables that have lead to the creation of special instructions. That is, variables that are live only inside one basic block are not taken into consideration. The figure emphasizes the difference between the conversion required by the two forward analyses and the SSI conversion. On the average, for each variable whose conversion is requested by either the ABCD or the CCP client, we will create 0.6 ϕ-functions, and 1.3 σ-functions. On the other hand, SSI will insert 6.1 σ-functions and 2.7 ϕ-functions per variable.

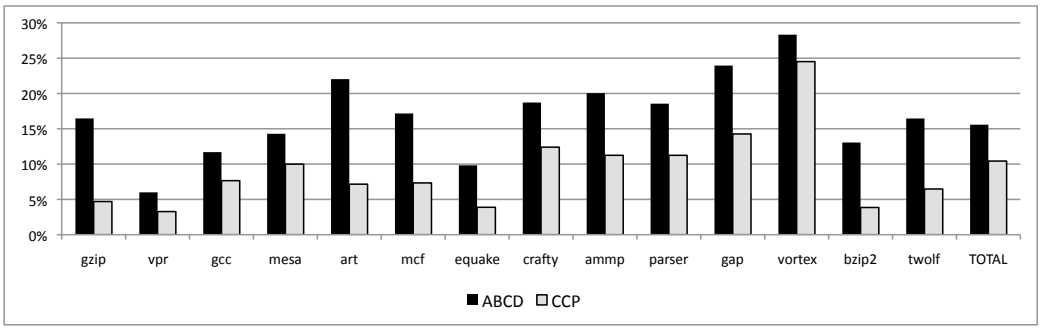

Figure 15: Number of ϕ and σ-functions produced by different live range splitting strategies. 100% is the number of instructions inserted by the SSI conversion.

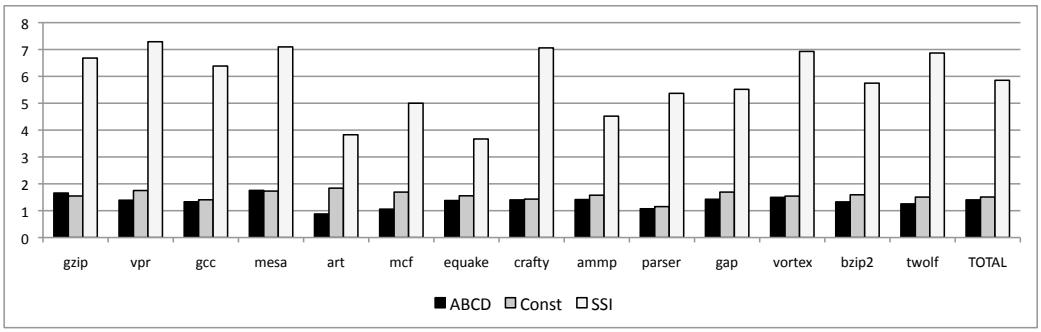

Figure 16: Average number of ϕ and σ-functions produced per variable.

Finally, Figure 17 outlines how much each live range splitting strategy increases program size. We show results only to the ABCD and CCP clients, to keep the chart easy to read. The SSI conversion increases program size in 17.6% on average. This is an absolute value, i.e., we sum up every ϕ and σ function inserted, and divide it by the number of bytecode instructions in the original program. This compiler already uses the SSA-form by default, and we do not count as new instructions the ϕ-functions originally used in the program. The ABCD client increases program size by 2.75%, and the CCP client increases program size by 1.84%.

An interesting question that deserves attention is "What is the benefit of using a sparse data-flow analysis in practice?" We have not implemented dense versions of the ABCD or the CCP clients. However, previous works have shown that sparse analyses tend to outperform equivalent dense versions

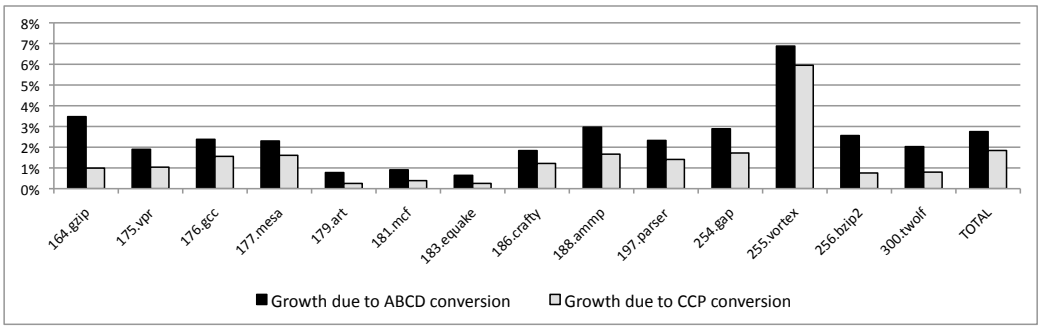

Figure 17: Growth in program size due to the insertion of new ϕ and σ functions to perform live range splitting.

in terms of time and space efficiency [21, 22]. In particular, the e-SSA format used by the ABCD and the CCP optimizations is the same program representation adopted by the tainted flow framework of Rimsa *et al.* [15], which has been shown to be faster than a dense implementation of the analysis, even taking the time to perform live range splitting into consideration.

5. Conclusion

This paper has presented a systematic way to build program representations that suit data-flow analyses. We build different program representations by splitting the live ranges of variables. The way in which we split live ranges depends on two factors. First, which program points produce new information, e.g., uses, definitions, tests, etc. Second, how this information propagates along the variable live range: forwardly or backwardly. We have used an implementation of our framework in LLVM to convert programs to the Static Single Information form [27], and to provide intermediate representations to the ABCD array bounds-check elimination algorithm [6] and to Wegman *et al.*'s Conditional Constant Propagation algorithm [20]. This very implementation has been used by Couto *et al.* [48] to provide the program representation required to implement Gawlitza *et al.*'s [40] range analysis algorithm. We have also used our live range splitting algorithm, implemented in the `phc` PHP compiler [49], to provide the Extended Static Single Assignment form necessary to solve the tainted flow problem [15].

Acknowledgments. This project has been made possible by the cooperation FAPEMIG-INRIA, grant 11/2009.

6. References

[1] R. T. Prosser, Applications of boolean matrices to the analysis of flow diagrams, in: Eastern joint IRE-AIEE-ACM computer conference, ACM, 1959, pp. 133–138.

[2] F. E. Allen, Control flow analysis, SIGPLAN Not. 5 (1970) 1–19.

[3] F. E. Allen, J. Cocke, A program data flow analysis procedure, Communications of the ACM 19 (1976) 137–146.

[4] G. A. Kildall, A unified approach to global program optimization, in: POPL, ACM, 1977, pp. 194–206.

[5] M. S. Hecht, Flow Analysis of Computer Programs, Elsevier, 1977.

[6] R. Bodik, R. Gupta, V. Sarkar, ABCD: eliminating array bounds checks on demand, in: PLDI, ACM, 2000, pp. 321–333.

[7] W. B. Ackerman, Efficient Implementation of Applicative Languages, Ph.D. thesis, MIT, 1984.

[8] R. Cartwright, M. Felleisen, The semantics of program dependence, SIGPLAN Not. 24 (1989) 13–27.

[9] L. Damas, R. Milner, Principal type-schemes for functional programs, in: POPL, ACM, New York, NY, USA, 1982, pp. 207–212.

[10] S. Tobin-Hochstadt, M. Felleisen, The design and implementation of typed scheme, POPL (2008) 395–406.

[11] R. Johnson, K. Pingali, Dependence-based program analysis, in: PLDI, ACM, 1993, pp. 78–89.

[12] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, T. Sherwood, Bitwidth cognizant architecture synthesis of custom hardware accelerators, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 20 (Nov 2001) 1355–1371.

[13] M. G. Nanda, S. Sinha, Accurate interprocedural null-dereference analysis for java, in: ICSE, pp. 133–143.

[14] J. B. Plevyak, Optimization of Object-Oriented and Concurrent Programs, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1996.

[15] A. A. Rimsa, M. D'Amorim, F. M. Q. Pereira, Tainted flow analysis on e-SSA-form programs, in: CC, Springer, 2011, pp. 124–143.

[16] S. Roy, Y. N. Srikant, The hot path ssa form: Extending the static single assignment form for speculative optimizations, in: CC, pp. 304–323.

[17] B. Scholz, C. Zhang, C. Cifuentes, User-input dependence analysis via graph reachability, Technical Report, Sun, Inc., 2008.

[18] M. Stephenson, J. Babb, S. Amarasinghe, Bidwidth analysis with application to silicon compilation, in: PLDI, ACM, 2000, pp. 108–120.

[19] Z. Su, D. Wagner, A class of polynomially solvable range constraints for interval analysis without widenings, Theoretical Computeter Science 345 (2005) 122–138.

[20] M. N. Wegman, F. K. Zadeck, Constant propagation with conditional branches, TOPLAS 13 (1991).

[21] J.-D. Choi, R. Cytron, J. Ferrante, Automatic construction of sparse data flow evaluation graphs, in: POPL, ACM, 1991, pp. 55–66.

[22] G. Ramalingam, On sparse evaluation representations, Theoretical Computer Science 277 (2002) 119–147.

[23] K. Pingali, G. Bilardi, APT: A data structure for optimal control dependence computation, in: PLDI, ACM, 1995, pp. 211–222.

[24] K. Pingali, G. Bilardi, Optimal control dependence computation and the roman chariots problem, in: TOPLAS, ACM, 1997, pp. 462–491.

[25] R. Johnson, D. Pearson, K. Pingali, The program tree structure, in: PLDI, ACM, 1994, pp. 171–185.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, TOPLAS 13 (1991) 451–490.

[27] S. Ananian, The Static Single Information Form, Master's thesis, MIT, 1999.

[28] J. Singer, Static Program Analysis Based on Virtual Register Renaming, Ph.D. thesis, University of Cambridge, 2006.

[29] B. Boissinot, A. Darte, F. Rastello, B. D. de Dinechin, C. Guillon, Revisiting out-of-SSA translation for correctness, code quality, and efficienty, in: CGO, IEEE, 2009, pp. 114–125.

[30] L. George, B. Matthias, Taming the ixp network processor, in: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLDI'03), ACM, 2003, pp. 26–37.

[31] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, P. Tu, Register promotion by sparse partial redundancy elimination of loads and stores, in: Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation (PLDI'98), ACM, 1998, pp. 26–37.

[32] F. K. Zadeck, Incremental Data Flow Analysis in a Structured Program Editor, Ph.D. thesis, Rice University, 1984.

[33] B. Boissinot, S. Hack, D. Grund, B. D. de Dinechin, F. Rastello, Fast liveness checking for SSA-form programs, in: CGO, IEEE, 2008, pp. 35–44.

[34] C. Lattner, V. S. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: CGO, IEEE, 2004, pp. 75–88.

[35] F. Nielson, H. R. Nielson, C. Hankin, Principles of program analysis, Springer, 2005.

[36] E. Duesterwald, R. Gupta, M. L. Soffa, Reducing the cost of data flow analysis by congruence partitioning, in: Compiler Construction, 5th International Conference (CC'94), volume 786 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 357–373.

[37] A. Miné, The octagon abstract domain, Higher Order Symbol. Comput. 19 (2006) 31–100.

[38] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, Cambridge University Press, 2nd edition, 2002.

[39] C. Chambers, D. Ungar, Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language, SIGPLAN Not. 24 (1989) 146–160.

[40] T. Gawlitza, J. Leroux, J. Reineke, H. Seidl, G. Sutre, R. Wilhelm, Polynomial precise interval analysis revisited, Efficient Algorithms 1 (2009) 422 – 437.

[41] J. hoon An, A. Chaudhuri, J. S. Foster, M. Hicks, Dynamic inference of static types for ruby, in: POPL, ACM, 2011, pp. 459–472.

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, An efficient method of computing static single assignment form, in: POPL, pp. 25–35.

[43] M. Weiss, The transitive closure of control dependence: the iterated join, TOPLAS 1 (1992) 178–190.

[44] P. Briggs, K. D. Cooper, L. Torczon, Improvements to graph coloring register allocation, TOPLAS 16 (1994) 428–455.

[45] F. M. Q. Pereira, J. Palsberg, SSA elimination after register allocation, in: CC, pp. 158 – 173.

[46] Z. Su, D. Wagner, A class of polynomially solvable range constraints for interval analysis without widenings and narrowings, in: TACAS, pp. 280–295.

[47] R. Barik, C. Grothoff, R. Gupta, V. Pandit, R. Udupa, Optimal bitwise register allocation using integer linear programming., in: LCPC, volume 4382 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 267–282.

[48] D. do Couto Teixeira, F. M. Q. Pereira, The design and implementation of a non-iterative range analysis algorithm on a production compiler, in: SBLP, SBC, 2011, pp. 45–59.

[49] P. Biggar, Design and Implementation of an Ahead-of-Time Compiler for PHP, Ph.D. thesis, Trinity College Dublin, 2009.

[50] V. C. Sreedhar, R. D. ching Ju, D. M. Gillies, V. Santhanam, Translating out of static single assignment form, in: SAS, Springer-Verlag, 1999, pp. 194–210.

[51] Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, S. W. Reeves, Fast copy coalescing and live-range identification, in: PLDI, ACM, 2002, pp. 25–32.

Appendix A. Isomorphism to Sparse Evaluation Graphs

Given a control flow graph G, Choi *et al.* define a sparse evaluation graph as a tuple $\langle N_{SG}, E_{SG}, M \rangle$, such that:

- N_{SG} is a set of nodes defined as follows:
  1. N_{SG} contains a node n_s representing the entry point $s \in G$;
  2. N_{SG} contains a node n_p for each point $p \in G$ that is associated with a non-identity transfer function.
  3. N_{SG} contains a node n_m for each point m in the iterated dominance frontier of the points of G used to build the nodes in step (1) and (2). These are called *meet* nodes.

- We let P denote the set of points $p \in G$ used in step 2 above, plus the point $s \in G$ used in step 1 above; we let M denote the set of points $m \in G$ used in step 3 above; if we let $S = P \bigcup M$ then we define E_{SG} as follows:
  1. there is an edge $(n_q, n_m) \in N_{SG}^2$ whenever $m \in M$ and q is, among all the nodes in S, the immediate dominator of one of the CFG predecessors of m. See `search(3b)` and `link(2b)` in Choi *et al* [21];
  2. there is an edge $(n_q, n_p) \in N_{SG}^2$ whenever $p \in P$, and q is, among all the nodes in S, the immediate dominator of p. See `search(1)` and `link(2b)` [21];

- The mapping function $M : E_G \mapsto N_{SG}$ associates to each edge (u, v) of the CFG the node $n_q \in N_{SG}$, whenever $q \in S$ is the immediate dominator of $u \in G$. See `search(3a)` [21]. This is done through the recursive function `search` that performs a topological traversal of the CFG (DFS of the dominance tree; See `search(4)` [21]).

Theorem 2 states that, for forward partitioned variable data-flow problems (PVP), the algorithm in Figure 8 can build program representations isomorphic to Sparse Evaluation Graphs. The proof that this result holds for backward data-flow problems, is analogous, and we omit it.

Lemma 1 (CFG cover). *Let $Prog$ be a program with its corresponding CFG G with start node s, and exit node x. Let $Prog'$ be the program that we obtain from $Prog$ by:*

1. *adding a pseudo-definition of each variable to s;*
2. *adding a pseudo-use of each variable to x;*
3. *placing a pseudo-use of a variable v at each point where v is defined;*
4. *converting the resulting program into SSA form.*

If v is a variable in $Prog$, then the live ranges of the different names of v in $Prog'$ completely partition the program points of G. In other words, each program point of G belongs to exactly one live range of v in $Prog'$.

Proof. First, v is alive at every point of G, due to transformations (1), (2) and (3). Therefore, if V is the set of the different names of v after the conversion to SSA form in step (4), then any program point of G belongs to the live range of at least one $v' \in V$. The result follows from a well-know property of Cytron's SSA-form conversion algorithm [26], which, as observed by Sreedhar *et al.* [50], creates variables with non-intersecting live ranges. In other words, after the SSA renaming, two different names of v cannot be simultaneously alive at a program point p.

Theorem 2 (Equivalence SSI/SEG). *Given a forward Sparse Evaluation Graph (SEG) that represents a variable v in a program representation $Prog$ with CFG G, there exits a live range splitting strategy that once applied on v builds a program representation that is isomorphic to SEG.*

Proof. We argue that the SEG of v is isomorphic to the representation of v in $Prog'$, the program representation that we derive from $Prog$ by applying the transformations 1-3 listed in Lemma 1 in addition to a pass of SSIfy. If we let P, as before, be defined as the set of CFG points associated with non-identity transfer functions, plus the start node s of the CFG, then after we apply the splitting strategy $P_{\downarrow}$, we have that:

1. there will be exactly one definition per node of P and one definition per node of $DF^+(P)$. So there is an one-to-one correspondence between SSA definitions and SG nodes.
2. From Lemma 1 the live-ranges of the different names of v provides a partitioning of the points of G. If v' is a new name of v, then each

program point where v' is alive is dominated by v''s definition[5]. Each program point belongs to the live-range of the name of v whose definition immediately dominates it (among all definitions). Thus, live ranges give origin to a function that maps SSA definitions to program points. Consequently, there is an isomorphism between the live-ranges and the mapping function M.

3. def-use chains on $Prog'$ are isomorphic to the edges in E_{SG}: indeed a SEG node n_p is linked to n_q whenever (i) n_p immediately dominates n_q if $q \in P$; or (ii) n_q is in the dominance frontier of n_p if $q \in M$. In the former case the definition of v at p reaches the (pseudo-)use of v at q. In the latter this definition reaches the use of v at the ϕ-function placed at q by $\mathsf{SSIfy}(v, P_{\downarrow})$.

In the proof of Theorem 2 we had to augment the program with a pseudo-definition of v at the CFG's entry point and a pseudo-use at every actual definition of v and at the CFG's exit point. The difference between a code with or without pseudo uses/defs is related to the necessity to compute data-flow information beyond the live-ranges of variables or not. This necessity exists for optimizations such as partial redundancy elimination, which may move, create or delete code.

Figure A.18 compares SEG and the forward live range splitting strategy in the example taken from Figure 11 of Choi *et al.* [21], which shows the reaching uses analysis. In the left we see the original program, and in the middle the SEG built for a forward flow analysis that extracts information from uses of variables. We have augmented the edges in the left CFG with the mapping M of SEG nodes to CFG edges. In the right we see the same CFG, augmented with pseudo defs and uses, after been transformed by SSIfy applied on the points $\{S, 4, 5, 7, 11, 12\}_{\downarrow}$. The edges of this CFG are labeled with the definitions of v live there.

Appendix B. Correctness of our SSIfication

In this section we consider a unidirectional forward (resp. backward) PVL problem stated as a set of equations $[v]^i = [v]^i \wedge F_v^s(\ldots)$ for every variable v, each program point i, and each $s \in \mathit{pred}(i)$ (resp. $s \in \mathit{succ}(i)$). We rely on the concepts introduced by Definition 2 in order to prove Theorem 4.

[5]This is a classical result of SSA-form. See Budimlic *et al.* [51] for a proof

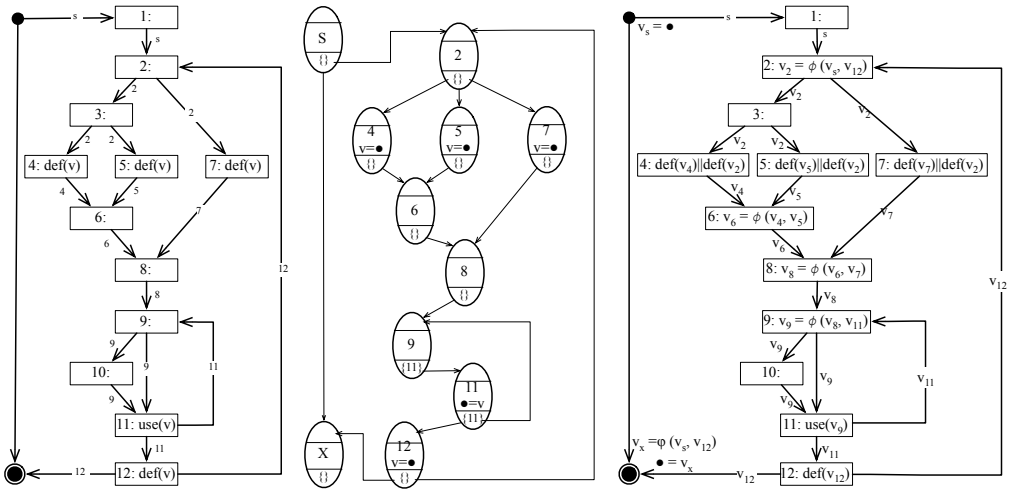

Figure A.18: Example of equivalence between SEGs and our live range splitting strategy for reaching uses.

Definition 2 (Trivial/constant transfer functions. Dependencies). Let $\mathcal{L}_{v_1} \times \mathcal{L}_{v_2} \times \ldots \times \mathcal{L}_{v_n}$ be the decomposition per variable of lattice $\mathcal{L}$, where $\mathcal{L}_{v_i}$ is the lattice associated with variable v_i. Let F_v be a transfer function from $\mathcal{L}$ to $\mathcal{L}_v$. We say that F_v is *trivial* if:

$$\forall x = ([v_1], \ldots, [v_n]) \in \mathcal{L},\ F_v(x) = x_v$$

We say that F_v is *constant with value* $C \in \mathcal{L}$ if:

$$\forall x \in \mathcal{L},\ F_v(x) = C$$

If F_v is constant with value $\perp$, e.g., $F_v(x) = \perp$, then we say that F_v is *undefined*. Finally, we say that F_v *depends on variable* v_j if:

$$\begin{array}{l}\exists x = ([v_1], \ldots, [v_n]) \neq ([v_1]', \ldots, [v_n]') = x' \text{ in } \mathcal{L} \\ \text{such that } [\forall k \neq j,\ [v_k] = [v_k]' \wedge F_i(x) \neq F_i(x')]\end{array}$$

Lemma 2 (Live range preservation). *If variable v is live at a program point i, then there is a version of v live at i after we run* SSIfy*.*

Proof. Split cannot remove any live range of v, as it only inserts "copies" from v to v, e.g., each copy has the same source and destination. Rename

removes live ranges of v, but it replaces them with the live ranges of new versions of this variable whenever a use of v is renamed. Clean only removes "copies"; hence, all the original instructions remain in the code. □

Lemma 3 (Non-Overlapping). *Two different versions of v, e.g., v_i and v_j cannot be live at a program point i transformed by* SSIfy.

Proof. The only algorithm that creates new versions of v is rename. Each new version of v is unique, as we ensure in line 27-29 of the algorithm. If rename changes the use of v to v_i at i, then there exists a definition of v_i at some program point i' that dominates i, as we ensure in line 22 of the algorithm. Lets assume that we have two versions of v, e.g., v_i and v_j, live at a program point i, in order to derive a contradiction. in this case, there exist program points i_i where v_i is used, and i_j where v_j is used, reachable from i. And exist a program point i'_i where v_i is defined, and a program point i'_j where v_j is defined, so that i'_i dominates i_i, and i'_j dominates i_j. Now, if neither i'_i dominates i'_j nor vice-versa, then we have a contradiction, because, given that i'_i reaches i_j and i'_j reaches i_i, then neither i'_i would dominates i_i, nor i'_j would dominates i_j. Without loss of generality, lets assume that i'_i dominates i'_j. in this case, rename visits i'_i first, and upon visiting i'_j, places the definition of v_j on top of the definition of v_i in the stack in line 30. Thus, i'_j cannot dominate i_i, or we would have, at that program point, a use of v_j, instead of v_i. In this case, i_j is live past the dominance frontier of i'_i, forcing split (line 14) to create a ϕ-function that dominates i_i, at a program point that is dominated by i'_i; hence, creating a new definition v_ϕ of v. Therefore, at i_i we would have a use of v_ϕ instead of v. □

Theorem 3 (Semantics). SSIfy *maintains the following property: if a value n written to variable v at program point i' is read at a program point i in the original program, then the same value assigned to a version of variable v at program point i' is read at a program point i after transformation.*

Proof. For simplicity, we will extend the meaning of "copy" to include not only the parallel copies placed at interior nodes, but also ϕ and σ-functions. Split cannot create new values, as it only inserts "copies". Clean cannot remove values, as it only removes "copies". From the hypothesis we know that the definition of v that reaches i is live at i. From Lemma 2 we know that there is a version of v live at i. From Lemma 3 we know that only one version of v can be live at i, and so rename cannot send new values to i. □

Now suppose that the program, not necessarily under SSI form, fulfills INFO and LINK as defined in Property 2 for a system of monotone equations E_{dense}, given as a set of constraints $[v]^i \sqsubseteq F_v^s([v_1]^s, \ldots, [v_n]^s)$. Consider a live range splitting strategy $\mathcal{P}_v$ that *includes* for each variable v the set of program points $I_\downarrow$ (resp. $I_\uparrow$) where F_v^s is non-trivial. The following theorem states that Algorithm SSIfy creates a program form that fulfills the Static Single Information property.

Theorem 4 (Correctness of SSIfy). *Given the conditions stated above, Algorithm SSIfy(v, $\mathcal{P}_v$) creates a new program representation such that:*

1. *there exists a system of equations E_{dense}^{ssi}, isomorphic to E_{dense} for which the new program representation fulfills the SSI property.*
2. *if E_{dense} is monotone then E_{dense}^{ssi} is also monotone.*

Proof. We derive from this new program representation a system of equations isomorphic to the initial one by associating trivial transfer functions with the newly created "copies". The INFO and LINK properties are trivially maintained. As only trivial and constant functions have been added, monotonicity is maintained.

To show that we provide SPLIT, we must first show that each $i \in \text{live}(v)$ where F_v^s is non-trivial contains a definition (resp. last use) of v. The function split separates these points in lines 9 and 16, and later, in line 23, inserts definitions in those points. Second, we must show that each join (resp. split) node for which E_{dense} has possibly different values on its incoming edges should have a ϕ-function (resp. σ-function) for v. These points are separated in lines 7 and 14 of split. To see why this is the case, notice that line 7 separates the points in the iterated dominance frontier of points that originate information that flows forward. These are, as a direct consequence of the definition of iterated dominance frontier, the points where information collide. Similarly, line 14 separates the points in the post-dominance frontier of regions which originate information that flows backwardly.

We ensure VERSION as a consequence of the SSA conversion. All our program representations preserve the SSA representation, as we include the definition sites of v in line 11 of split. Function rename ensures the existence of only one definition of each variable in the program code (line 27), and that each definition dominates all its uses (consequence of the traversal order). Therefore, the newly created live ranges are connected on the dominance tree of the source program. Function rename also creates a new program

representation for which it is straightforward to build a system of equations E^{ssi}_{dense} isomorphic to E_{dense}: Firstly, the constraint variables are renamed in the same way that program variables are. Secondly, for each program variable, new system variables bound to $\perp$ are created for each program point outside of its live-range. □

Appendix C. Equivalence between sparse and dense analyses.

We have shown that SSIfy transforms a program P into another program P^{ssi} with the same semantics. Furthermore, this representation provides the SSI property for a system of equations E^{ssi}_{dense} that we extract from P^{ssi}. This system is isomorphic to the system of equations E_{dense} that we extract from P. From the so obtained program under SSI for the constrained system E^{ssi}_{dense}, Definition 1 shows how to construct a sparse constrained system E^{ssi}_{sparse}. When transfer functions are monotone and the lattice has finite height, Theorem 1 states the equivalence between the sparse and the dense systems. The purpose of this section is to prove this theorem. We start by introducing the notion of *coalescing*. Let E be a constraint system that associates with each $1 \leq i \leq n$ the constraint $a_i \sqsubseteq H_i(a_1, \ldots, a_n)$, where each a_i is an element of a lattice $\mathcal{L}$ of finite height, and H_i is a monotone function from $\mathcal{L}^n$ to $\mathcal{L}$. Let $(A_1, \ldots, A_n)$ be the maximum solution to this system, and let $1 \leq m \leq n$ such that $\forall i,\ 1 \leq i \leq m,\ A_i = A_m$. We define a "coalesced" constraint system E_{coal} in the following way: for each $1 \leq i \leq m$ we create the constraint $b_m \sqsubseteq H_i(b_m, \ldots, b_m, b_{m+1}, \ldots, b_n)$; for each $m < i \leq n$ we create the constraint $b_i \sqsubseteq H_i(b_m, \ldots, b_m, b_{m+1}, \ldots, b_n)$. Lemma 4 shows that coalescing preserves the maximum solution of the original system.

Lemma 4 (Equivalence with coalescing). *If E is a constraint system with maximum solution $(A_1, \ldots, A_m, \ldots, A_n)$, for any $i, j, 1 \leq i, j \leq m$ we have that $A_i = A_j$, and E_{coal} is the "coalesced" system that we derive from E, then the maximum solution of E_{coal} is $(A_m, \ldots, A_n)$.*

Proof. Both system have a (unique) maximum solution (see e.g. [35]), although the solution of the "coalesced" system has smaller cardinality, e.g., n-m+1. Now, as $(A_m, \ldots, A_m, A_{m+1}, \ldots, A_n)$ is a solution to E, by definition of E_{coal}, $(A_m, \ldots, A_n)$ is a solution to E_{coal}. Let us prove that this solution is maximum, i.e. for any solution $(B_m, \ldots, B_n)$ of E_{coal}, we have $(B_m, \ldots, B_n) \sqsubseteq (A_m, \ldots, A_n)$. By definition of E_{coal}, we have that

$(B_m, \ldots, B_m, B_{m+1}, \ldots, B_n)$ is a solution to E. As $(A_1, \ldots, A_n)$ is maximum, we have $(B_m, \ldots, B_m,\ B_{m+1}, \ldots, B_n) \sqsubseteq (A_1, \ldots, A_n)$. So $(B_m, \ldots, B_n) \sqsubseteq (A_m, \ldots, A_n)$. $\square$

We now prove Theorem 1, which states that there exists a direct mapping between the maximum solution of a dense constraint system associated with a SSI-form program, and the sparse system that we can derive from it, according to Definition 1.

PROOF. The constraint systems E^{ssi}_{dense} and E^{ssi}_{sparse} have a maximum unique solution, because the transfer functions are monotone and $\mathcal{L}$ has finite height

The idea of the proof is to modify the constraint system E^{ssi}_{dense} into a system equivalent to E^{ssi}_{sparse}. To accomplish this transformation, we (i) replace each F^s_v by G^s_v, where G^s_v is constructed as in Definition 1; (ii) for each v, coalesce $[v]^i_{i \in live(v)}$ into $[v]$; (iii) coalesce all other constraint variables into $[v_\perp]$.

The LINK property allows us to replace F^s_v by G^s_v. Due to SPLIT, a new variable is defined at each point where information is generated, and due to VERSION there is only one live range associated with each variable. Hence, $([v]^i)_{i \in live(v)}$ is invariant. Due to INFO, we have that $([v]^i)_{i \notin live(v)}$ is bound to $\perp$. Due to Lemma 4, we know that this new constraint system has a maximum solution $(Y_v)_{v \in variables \cup \perp}$: X^i_v equals Y_v for all $i \in live(v)$, and $Y_\perp$ otherwise.

We translate each constraint $[v]^i \sqsubseteq F^s_v([v_1]^s, \ldots, [v_n]^s)$, in the original system, to a constraint in the "coalesced" one in the following way:

$$\begin{cases} \text{if } i \in live(v): & \text{if } s \in defs(v) & : [v] \sqsubseteq G^s_v([a], \ldots, [b]) & (1) \\ & \text{else} & : [v] \sqsubseteq [v] & (2) \\ \text{otherwise} & & : [v_\perp] \sqsubseteq \perp & (3) \end{cases}$$

Case (1) follows from LINK, case (2) follows from SPLIT, and case (3) follows from INFO. By ignoring $y_\perp$ that appears only in (3), and by removing the constraints produced by (2), which are useless, we obtain E^{ssi}_{sparse}.