Global Sparse Analysis Framework

HAKJOO OH, KIHONG HEO, WONCHAN LEE, WOOSUK LEE, DAEJUN PARK, JEEHOON KANG, and KWANGKEUN YI, Seoul National University

In this article, we present a general method for achieving global static analyzers that are precise and sound, yet also scalable. Our method, on top of the abstract interpretation framework, is a general sparse analysis technique that supports relational as well as nonrelational semantics properties for various programming languages. Analysis designers first use the abstract interpretation framework to have a global and correct static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, analysis designers add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our method prescribes what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

We formally present our framework and show that existing sparse analyses are all restricted instances of our framework. In addition, we show more semantically elaborate design examples of sparse nonrelational and relational static analyses. We then present their implementation results that scale to globally analyze up to one million lines of C programs. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.

Categories and Subject Descriptors: F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms: Programming Languages, Program Analysis

Additional Key Words and Phrases: Static analysis, abstract interpretation, sparse analysis

ACM Reference Format:
Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global sparse analysis framework. ACM Trans. Program. Lang. Syst. 36, 3, Article 8 (September 2014), 44 pages.
DOI: http://dx.doi.org/10.1145/2590811

1. INTRODUCTION

Precise, sound, scalable yet global static analyzers have been unachievable in general. Other than almost syntactic properties, once the target property becomes slightly deep in semantics, it has been a daunting challenge to achieve the four goals in a single static analyzer. This situation explains why, for example, in the static error detection tools for full C, there exists a clear dichotomy: either "bug-finders" that risk being unsound yet scalable or "verifiers" that risk being unscalable yet sound. No such tools are scalable to globally analyze million lines of C code while being sound and precise enough for practical use.

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP)/National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609).
Author's addresses: H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi, Department of Computer Science and Engineering, Seoul National University, 1 Kwanak-ro Kwanak-gu, Seoul, Korea.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 0164-0925/2014/09-ART8 $15.00
DOI: http://dx.doi.org/10.1145/2590811

In this article, we present a general method for achieving global static analyzers that are precise and sound, yet also scalable. Our approach generalizes the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpretation framework [Cousot and Cousot 1977, 1979, 1992] guides us to design sound yet arbitrarily precise static analyzers for any target language, analysis designers first use the framework to have a global and correct static analyzer whose scalability is unattended. Upon this underlying sound static analyzer, analysis designers add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework prescribes what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

Our framework bridges the gap between the two existing technologies—abstract interpretation and sparse analysis—toward the design of sound yet scalable global static analyzers. Note that although the abstract interpretation framework [Cousot and Cousot 1977, 1979, 1992] provides a theoretical knob to control the analysis precision without violating its correctness, the framework does not provide a knob to control the resulting analyzer's scalability while preserving its precision. On the other hand, existing sparse analysis techniques [Reif and Lewis 1977; Wegman and Zadeck 1991; Dhamdhere et al. 1992; Chase et al. 1990; Tok et al. 2006; Cytron and Ferrante 1995; Johnson and Pingali 1993; Hardekopf and Lin 2009, 2011] achieve scalability, but they are mostly algorithmic and tightly coupled with particular analyses. The sparse techniques are not general enough to be used for an arbitrarily complicated semantic analysis. A few techniques [Choi et al. 1991; Ramalingam 2002] are in general settings, but instead they take a coarse-grained approach to sparsity, where the entire (nonsparse) abstract state is propagated as a unit from program point to program point.

We formally present our framework, in which existing sparse analyses are all restricted instances of our framework. We show more semantically elaborate design examples of sparse nonrelational and relational static analyses. Then, we present their implementation results that scale to analyze up to one million lines of C programs.

Contributions. In this article, which is an extended version of Oh et al. [2012], we make the following contributions:

—We present a general framework for designing sparse static analysis. Given a baseline analysis defined by abstract interpretation, our framework provides a semantics-based method to define a sparse version that preserves the precision of the original analysis. On the theoretical side, our framework is general in three ways:
  —It is applicable to static analysis for various programming languages (e.g., imperative languages, functional languages).
  —It is applicable to static analysis with various semantics properties (e.g., arbitrary nonrelational analysis, packed relational analysis).
  —It is applicable to arbitrary trace partitioning (e.g., context-sensitivity, path-sensitivity, loop unrolling).

—We present new notions of data dependencies and their safe approximations, which is the key to the precision-preserving sparse analysis. Unlike conventional def-use chains, sparse analysis with our data dependency preserves the precision of the original nonsparse version.

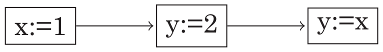
—We demonstrate the practicality of our framework for a realistic C static analyzer. Using our framework, we have derived a sparse version of Sparrow [Jhee et al. 2008; Oh and Yi 2010, 2011; Oh et al. 2011, 2012], an industrial-strength static analyzer for C programs. The sparse version is able to analyze programs up to 1 million lines with interval domain and up to 100K lines with packed octagon domain.

Outline. Section 2 gives an informal overview on sparse analysis. Section 3 presents our sparse analysis framework. Sections 4 and 5 design sparse nonrelational and relational analyses, respectively, based on our framework. Section 6 discusses several issues involved in the implementations. Section 7 presents the experimental studies. Section 8 discusses related work, and Section 9 concludes the article.

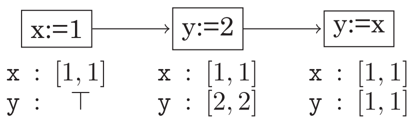
2. INFORMAL OVERVIEW ON SPARSE ANALYSIS

Before delving into our sparse analysis framework, we briefly give the idea of sparse analysis. Readers who are familiar with the notion of sparse analysis may wish to skip this overview and directly go to the next section.

Consider the following simple program, represented by a control flow graph:



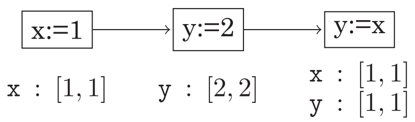
Suppose that we analyze the program with the interval domain. A conventional non-sparse static analysis would work by following the control flows of the program, resulting in the analysis results at each program point as follows (assume that x and y initially have ⊤):



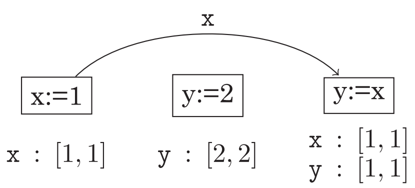
The results show that, for instance, x has interval value $[1, 1]$ and y has $[2, 2]$ right after the second statement.

Sparse analysis aims to optimize this conventional static analysis based on two observations:

—*Spatial sparsity (lightweight state)*: First, in the analysis of each statement, only a small part of the state is actually used. For example, in the first statement of the preceding program, the value of x is necessary to analyze the first statement but y is not. So, at each program point, sparse analysis stores only the values that are used in the analysis:



—*Temporal sparsity (express flow)*: The second observation is that the semantic dependencies among statements are usually sparse. For instance, the value of x in the first statement is not used at the next statement but is used at the last statement. Sparse analysis propagates abstract values along the semantic dependencies of the program rather than along control flows:



The label x on the dependency arrow means that the target statement has a dependency on x with the source statement.

Our sparse analysis framework in Section 3 provides a general theory for designing such a sparse analysis *without* violating the correctness and precision of the original nonsparse analysis.

3. SPARSE ANALYSIS FRAMEWORK

In this section, we develop our sparse analysis framework. Given a static analysis designed by abstract interpretation, our framework prescribes how to transform the analysis into its sparse version without violating the soundness and precision of the analysis. In Sections 3.2 and 3.3, we define the collecting semantics of the program. In Section 3.4, we specify a family of baseline abstractions that our framework considers. Then, we derive the sparse version of the baseline analysis in the remaining sections.

Although we use the C language as a main target in our experiments, we do not restrict the language. Our sparse analysis framework is general and applicable to various programming languages (functional, object oriented, etc), once their semantics are operationally defined as transition systems.

3.1. Notation

We write $\mathcal{P}(S)$ for the power set of S. Given partial function $f \in A \to B$, we write $\mathsf{dom}(f)$ for the domain of f. We write $f|_C$ for the restriction of function f to the domain $\mathsf{dom}(f) \cap C$ such that $f|_C(x) = f(x)$ if $x \in \mathsf{dom}(f) \cap C$ and $\perp$ otherwise. We write $f\backslash_C$ for the restriction of f to the domain $\mathsf{dom}(f)\backslash C$. We abuse the notation $f|_a$ and $f\backslash_a$ for the domain restrictions on singleton set $\mathsf{dom}(f) \cap \{a\}$ and $\mathsf{dom}(f)\backslash\{a\}$, respectively. We write $f[a \mapsto b]$ for function f with the value of a replaced by b. We write $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. For all domains, we assume appropriate $\perp$ and $\top$ as well as order $\sqsubseteq$ and join $\sqcup$. In particular, we define $\sqcup, \sqsubseteq, \top, \perp$ for functions in a pointwise fashion (e.g., $f \sqcup g = \lambda x. f(x) \sqcup g(x)$). We write $f[\{a_1, \ldots, a_n\} \overset{w}{\mapsto} b]$ for $f[a_1 \mapsto f(a_1) \sqcup b, \ldots, a_n \mapsto f(a_n) \sqcup b]$ (weak update).

Given a (potentially infinite) set S, we write S^+ for the set of all finite nonempty sequences of elements of S. When σ is a finite sequence, σ_k denotes the $(k+1)$th element of the sequence, σ_0 the first element, and $\sigma_\dashv$ the last element. Given a sequence $\sigma \in S^+$ and an element $s \in S$, $\sigma \cdot s$ denotes a sequence obtained by appending s to σ.

3.2. Programs

We describe a program's semantics as a transition system $(\mathbb{S}, \to, \mathbb{S}_\iota)$, where $\mathbb{S}$ is the set of states of the program, $(\to) \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation describing how the program execution progresses from one state to the next state, and $\mathbb{S}_\iota \subseteq \mathbb{S}$ denotes the set of initial states. A sequence σ of states is said to be a *trace* if σ is a (partial) execution sequence (i.e., $\sigma_0 \in \mathbb{S}_\iota \land \forall k. \sigma_k \to \sigma_{k+1}$). We abuse the notion of transition relation $\to$ for traces (i.e., $\sigma' \to \sigma \iff \exists s. (\sigma = \sigma' \cdot s) \land (\sigma'_\dashv \to s)$).

3.3. Collecting Semantics

The collecting semantics $[\![P]\!] \in \mathcal{P}(\mathbb{S}^+)$ of program P is the set of all finite traces of P:

$$[\![P]\!] = \{\sigma \in \mathbb{S}^+ \mid \sigma_0 \in \mathbb{S}_\iota \land \forall k. \sigma_k \to \sigma_{k+1}\}.$$

Note that the semantics $[\![P]\!]$ is the least fixpoint of the semantic function $F \in \mathcal{P}(\mathbb{S}^+) \to \mathcal{P}(\mathbb{S}^+)$ (i.e., $[\![P]\!] = \mathbf{lfp} F$), defined as follows:

$$F(\Sigma) = \mathbb{S}_\iota \cup \{\sigma \cdot s \mid \sigma \in \Sigma \land \sigma_\dashv \to s\}.$$

3.4. Baseline Abstraction

We abstract the collecting semantics of program P by the following Galois connections:

$$\mathcal{P}(\mathbb{S}^+) \xleftrightarrow[\alpha_1]{\gamma_1} \Delta \to \mathcal{P}(\mathbb{S}^+) \xleftrightarrow[\alpha_2]{\gamma_2} \Delta \to \hat{\mathbb{S}}.$$

The abstraction consists of two steps:

(1) *Partitioning abstraction* (α_1, γ_1): We abstract the set of traces $(\mathcal{P}(\mathbb{S}^+))$ into partitioned sets of traces $(\Delta \to \mathcal{P}(\mathbb{S}^+)$, where Δ is the set of partitioning indices).
(2) *State abstraction* (α_2, γ_2): For each partition, the associated set of traces is abstracted into an abstract state $(\hat{\mathbb{S}})$ that overapproximates the reachable states of the traces.

In Section 3.4.1, we specify the partitioning abstraction: the definitions of (α_1, γ_1) and semantic function $F^\pi \in (\Delta \to \mathcal{P}(\mathbb{S}^+)) \to (\Delta \to \mathcal{P}(\mathbb{S}^+))$. In Section 3.4.2, we will define the final abstract domain $(\Delta \to \hat{\mathbb{S}})$ and abstract semantic function $\hat{F} \in (\Delta \to \hat{\mathbb{S}}) \to (\Delta \to \hat{\mathbb{S}})$ as a further abstraction of the partitioning abstraction.

3.4.1. Partitioning Abstraction. Following Mauborgne and Rival [2005], we first partition the set of traces. Suppose that we are given a partitioning function $\pi : \Delta \to \mathcal{P}(\mathbb{S}^+)$ such that π is either a covering (i.e., $\mathbb{S}^+ = \bigcup_{i\in\Delta} \pi(i)$) or a partition (i.e., π is a covering and $\forall i, i' \in \Delta. i \neq i' \implies \pi(i) \cap \pi(i') = \emptyset$). Then, the following α_1 and γ_1

$$\alpha_1(\Sigma) = \lambda i \in \Delta. \Sigma \cap \pi(i)$$
$$\gamma_1(\phi) = \bigcup_{i\in\Delta} \phi(i)$$

form a Galois connection:

$$\mathcal{P}(\mathbb{S}^+) \xleftrightarrow[\alpha_1]{\gamma_1} \Delta \to \mathcal{P}(\mathbb{S}^+).$$

We define the semantic function $F^\pi \in (\Delta \to \mathcal{P}(\mathbb{S}^+)) \to (\Delta \to \mathcal{P}(\mathbb{S}^+))$ as follows:

$$F^\pi(\phi) = \lambda i \in \Delta. \alpha_1(\mathbb{S}_\iota)(i) \cup f_i\left(\bigcup_{i' \Rightarrow_\phi i} \phi(i')\right),$$

where $f_i \in \mathcal{P}(\mathbb{S}^+) \to \mathcal{P}(\mathbb{S}^+)$ is the semantic function for partitioning index i and $(\Rightarrow_\phi) \subseteq \Delta \times \Delta$ is the transition relation between partitioning indices.

Definition 3.1 (*Semantic Function*). Semantic function $f_i \in \mathcal{P}(\mathbb{S}^+) \to \mathcal{P}(\mathbb{S}^+)$ undertakes one step state transitions for index i:

$$f_i(\Sigma) = \{\sigma \mid \sigma' \in \Sigma \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}.$$

Given a set Σ of input traces, f_i makes their transitions one step forward if the resulting trace σ arrives at the current partitioning index i.

Definition 3.2 (*Transition Relation*). Transition relation $(\Rightarrow) \subseteq \Delta \times \Delta \times (\Delta \to \mathcal{P}(\mathbb{S}^+))$ is a ternary relation such that $i' \Rightarrow_\phi i$ indicates that one step transition in ϕ may happen from i' to i according to the partitioning function π:

$$(\Rightarrow) = \{(i', i, \phi) \mid \sigma' \in \phi(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}.$$

With this partitioning abstraction, we can design static analysis with general trace partitioning [Mauborgne and Rival 2005]. One of the conventional partitioning strategies is the control-point partitioning. Suppose that a state is decomposed into a control point in $\mathbb{C}$ and a memory state in $\mathbb{M}$ (i.e., $\mathbb{S} = \mathbb{C} \times \mathbb{M}$). We use $\mathbb{C}$ as the set of partitioning indices and let $\pi_{\mathbb{C}} \in \mathbb{C} \to \mathcal{P}(\mathbb{S}^+)$ partition $\mathbb{S}^+$ based on the final control point: $\pi_{\mathbb{C}}(c) = \{\sigma \in \mathbb{S}^+ \mid \exists m.\sigma_{\dashv} = (c, m)\}$. Then, the transition relation ($\Rightarrow$) denotes the control flows of the program, and f_i is the semantics function for control point i. The analysis with this partitioning is known as flow sensitive. Other partitioning strategies such as context sensitivity, path sensitivity, and loop unrolling are all specific instances of general trace partitioning.

The following lemma shows that the partitioning abstraction we designed earlier is indeed sound with respect to the collecting semantics.

LEMMA 3.3. $\alpha_1(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}F^{\pi}$.

PROOF. We prove $\alpha_1 \circ F \sqsubseteq F^{\pi} \circ \alpha_1$. Then, the soundness is obtained by the fixpoint transfer theorem [Cousot and Cousot 1977]:

$$\begin{array}{lll}
\forall \Sigma \in \mathcal{P}(\mathbb{S}^+), & & \\
(\alpha_1 \circ F)(\Sigma) & = \alpha_1(\mathbb{S}_\iota \cup \{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_{\dashv} \to s\}) & (\text{def. of } F) \\
& = \alpha_1(\mathbb{S}_\iota) \sqcup \alpha_1(\{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_{\dashv} \to s\}) & (\alpha_1 \text{ is distrib. over } \cup) \\
& = (\lambda i.(\alpha_1 \mathbb{S}_\iota)(i)) \sqcup (\lambda i.\{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_{\dashv} \to s\} \cap \pi(i)) & (\text{def. of } \alpha_1) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup \{\sigma \cdot s \in \pi(i) \mid \sigma \in \Sigma \wedge \sigma_{\dashv} \to s\} & (\text{def. of } \sqcup) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i(\Sigma) & (\text{def. of } f_i) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup \big(f_i(\Sigma) \cap f_i\big(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i')\big)\big) & (\text{Lemma 3.4}) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i\big(\Sigma \cap \big(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i')\big)\big) & (f_i \text{ is distrib. over } \cap) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i\big(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} (\Sigma \cap \pi(i'))\big) & (\text{set theory}) \\
& = \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i\big(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} (\alpha_1 \Sigma)(i')\big) & (\text{def. of } \alpha_1) \\
& = F^{\pi}(\alpha_1 \Sigma) & (\text{def. of } F^{\pi}) \\
& = (F^{\pi} \circ \alpha_1)(\Sigma). \quad \square &
\end{array}$$

LEMMA 3.4. $\forall \Sigma \subseteq \mathbb{S}^+, \forall i \in \Delta. f_i(\Sigma) \subseteq f_i(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i'))$.

PROOF. Note that $\bigcup_{i \in \Delta}(\alpha_1 \Sigma)(i) = \Sigma$ because π is a covering:

$$\begin{array}{rl}
\bigcup_{i \in \Delta}(\alpha_1 \Sigma)(i) & = \bigcup_{i \in \Delta}(\Sigma \cap \pi(i)) \\
& = \Sigma \cap \bigcup_{i \in \Delta} \pi(i) \\
& = \Sigma \cap \mathbb{S}^+ = \Sigma.
\end{array}$$

Now, $\forall \Sigma \subseteq \mathbb{S}^+$,

$$\begin{array}{rll}
f_i(\Sigma) & = \{\sigma \mid \sigma' \in \Sigma \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\} & (\text{def. of } f_i) \\
& = f_i(\{\sigma' \mid \sigma' \in \Sigma \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}) & (\text{def. of } f_i) \\
& = f_i\big(\{\sigma' \mid \sigma' \in \bigcup_{i' \in \Delta}(\alpha_1 \Sigma)(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}\big) & (\bigcup_{i' \in \Delta}(\alpha_1 \Sigma)(i') = \Sigma) \\
& = f_i\big(\bigcup_{i' \in \Delta}\{\sigma' \mid \sigma' \in (\alpha_1 \Sigma)(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}\big) & \\
& = f_i\big(\bigcup_{(i', i) \in \Delta \times \Delta}\{\sigma' \mid \sigma' \in (\alpha_1 \Sigma)(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}\big) & \\
& \subseteq f_i\big(\bigcup_{(i', i) \in \Delta \times \Delta \wedge \sigma' \in (\alpha_1 \Sigma)(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)} \pi(i')\big) & ((\alpha_1 \Sigma)(i') \subseteq \pi(i')) \\
& = f_i\big(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i')\big). & (\text{def. of } \Rightarrow) \quad \square
\end{array}$$

3.4.2. State Abstraction. Next, we abstract the partitioned collecting semantics by abstracting each partition's traces into an abstract state. Suppose that we have abstraction and concretization functions for set of traces (i.e., $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$) such that

$$\mathcal{P}(\mathbb{S}^+) \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$$

and $\alpha_{\mathbb{S}}$ is distributive over $\cup$ (i.e., $\forall \Sigma_1, \Sigma_2 \subseteq \mathbb{S}^+ . \alpha_{\mathbb{S}}(\Sigma_1 \cup \Sigma_2) = \alpha_{\mathbb{S}}(\Sigma_1) \sqcup \alpha_{\mathbb{S}}(\Sigma_2)$). Then, the state abstraction is defined as the following Galois connection:

$$\Delta \to \mathcal{P}(\mathbb{S}^+) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \Delta \to \hat{\mathbb{S}},$$

where α_2 and γ_2 are pointwise liftings of $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$, respectively. For instance,

$$\alpha_2(\phi) = \lambda i \in \Delta . \alpha_{\mathbb{S}}(\phi(i))$$
$$\gamma_2(\hat{\phi}) = \lambda i \in \Delta . \gamma_{\mathbb{S}}(\hat{\phi}(i)).$$

We consider a particular, yet general enough, family of abstract domains such that $\hat{\mathbb{S}}$ is of form $\hat{\mathbb{L}} \to \hat{\mathbb{V}}$, where $\hat{\mathbb{L}}$ is a finite set of abstract locations and $\hat{\mathbb{V}}$ is a (potentially infinite) set of abstract values. First, all nonrelational abstract domains, such as intervals [Cousot and Cousot 1977], are members of this family. Furthermore, the family also covers some numerical, relational domains. Practical relational analyses exploit *packed* relationality [Cousot et al. 2009; Miné 2006b; Venet and Brat 2004; Blanchet et al. 2003]; the abstract domain is of form $Packs \to \hat{\mathbb{R}}$, where *Packs* is a set of variable groups selected to be related together. $\hat{\mathbb{R}}$ denotes numerical constraints among variables in those groups. In such *packed* relational analysis, each variable pack is treated as an abstract location ($\hat{\mathbb{L}}$), and numerical constraints amount to abstract values ($\hat{\mathbb{V}}$). Example of the numerical constraints are the domains of octagons [Miné 2006b] and polyhedra[1] [Cousot and Halbwachs 1978].

The final abstract semantics is characterized as the least fixpoint of the following abstract semantic function $\hat{F} \in (\Delta \to \hat{\mathbb{S}}) \to (\Delta \to \hat{\mathbb{S}})$:

$$\hat{F}(\hat{\phi}) = \lambda i \in \Delta . \hat{f}_i \left(\bigsqcup_{i' \hookrightarrow_{\hat{\phi}} i} \hat{\phi}(i') \right), \tag{1}$$

where $\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ (Definition 3.5) is an abstract semantic function for partitioning index i and $(\hookrightarrow_{\hat{\phi}}) \subseteq \Delta \times \Delta$ (Definition 3.6) is an abstract transition relation.

Definition 3.5 (*Abstract Semantic Function*). Abstract semantic function $\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is an abstract counterpart of f_i, which satisfies the following conditions:

(1) $\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}} . \hat{s} \sqsubseteq \hat{s}' \Rightarrow \hat{f}_i(\hat{s}) \sqsubseteq \hat{f}_i(\hat{s}')$
(2) $\alpha_{\mathbb{S}} \circ f_i \sqsubseteq \hat{f}_i \circ \alpha_{\mathbb{S}}$
(3) $\forall \hat{s} \in \hat{\mathbb{S}} . \alpha_{\mathbb{S}}((\alpha_1 \mathbb{S}_\iota)(i)) \sqsubseteq \hat{f}_i(\hat{s})$.

The first condition says that $\hat{f}_i$ is monotone. The second and third conditions ensure the soundness of the abstract semantics. In particular, the third condition requires that $\hat{f}_i$ subsumes the initial traces. If $\hat{f}_i$ did not satisfy the third condition, we would have defined $\hat{F}$ as $\hat{F}(\hat{\phi}) = \lambda i \in \Delta . \alpha_{\mathbb{S}}((\alpha_1 \mathbb{S}_\iota)(i)) \sqcup \hat{f}_i(\bigsqcup_{i' \hookrightarrow_{\hat{\phi}} i} \hat{\phi}(i'))$, which defines fundamentally the same analysis as (1). We chose to subsume the initial traces by $\hat{f}_i$ because this makes our subsequent formalization simpler. Next, we define abstract transition relation $(\hookrightarrow_{\hat{\phi}}) \subseteq \Delta \times \Delta$ between partitioning indices.

Definition 3.6 (*Abstract Transition Relation*). Abstract transition relation $(\hookrightarrow) \subseteq \Delta \times \Delta \times (\Delta \to \hat{\mathbb{S}})$ is an abstract counterpart of $\Rightarrow$, which satisfies the following conditions:

[1]Some domains such as the convex polyhedra is not associated with a Galois connection. In this case, we could reformulate the analysis only assuming the existence of a concretization function.

(1) $(\hookrightarrow_{\alpha_2(\phi)}) \supseteq (\Rightarrow_\phi) = \{(i', i) \in \Delta \times \Delta \mid \sigma' \in \phi(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}$
(2) $\forall \hat{\phi}', \hat{\phi} \in \Delta \to \hat{\mathbb{S}}.\hat{\phi}' \sqsubseteq \hat{\phi} \implies (\hookrightarrow_{\hat{\phi}'}) \subseteq (\hookrightarrow_{\hat{\phi}})$
(3) $\forall \hat{\phi}', \hat{\phi} \in \Delta \to \hat{\mathbb{S}}, i', i \in \Delta.(\hat{\phi}'(i') = \hat{\phi}(i') \implies (i' \hookrightarrow_{\hat{\phi}'} i \iff i' \hookrightarrow_{\hat{\phi}} i))$.

The first condition means the soundness of $\hookrightarrow$. The second condition says that $\hookrightarrow$ is monotone on $\hat{\phi}$. The last condition means that the next index i of the current index i' is determined solely by the abstract state at i'.

Note that our baseline analysis is able to express a family of static analysis in which data and control are mutually dependent: $\hookrightarrow_{\hat{\phi}}$ (control) depends on $\hat{\phi}$ (data, analysis process' intermediate results), and the analysis process $\hat{F}$ is defined using $\hookrightarrow_{\hat{\phi}}$. It is not uncommon to design static analysis this way. For example, for functional languages or object-oriented languages, the desired precision is usually obtained only when control flow information is simultaneously computed during the analysis. On the other hand, in "C-like" languages, it is acceptable to assume that the control flow relation is fixed and available before the analysis. Our baseline analysis covers both cases.

The following lemmas show the soundness of the abstract semantics.

LEMMA 3.7. $\alpha_2(\mathbf{lfp}F^\pi) \sqsubseteq \mathbf{lfp}\hat{F}$.

PROOF. We prove $\alpha_2 \circ F^\pi \sqsubseteq \hat{F} \circ \alpha_2$. Then, the soundness is obtained by the fixpoint transfer theorem [Cousot and Cousot 1977]:

$$
\begin{array}{lll}
\forall \phi \in \Delta \to \mathcal{P}(\mathbb{S}^+), i \in \Delta, & & \\
(\alpha_2 \circ F^\pi)(\phi)(i) & = \alpha_{\mathbb{S}}\big((\alpha_1 \mathbb{S}_\iota)(i) \sqcup f_i\big(\bigcup_{i' \Rightarrow_\phi i} \phi(i')\big)\big) & \text{(def. of } \alpha_2 \text{ and } F^\pi) \\
& = \alpha_{\mathbb{S}}((\alpha_1 \mathbb{S}_\iota)(i)) \sqcup (\alpha_{\mathbb{S}} \circ f_i)\big(\bigcup_{i' \Rightarrow_\phi i} \phi(i')\big) & (\alpha_{\mathbb{S}} \text{ is distributive}) \\
& \sqsubseteq (\hat{f}_i \circ \alpha_{\mathbb{S}})\big(\bigcup_{i' \Rightarrow_\phi i} \phi(i')\big) & \text{((2) and (3) of Def. 3.5)} \\
& = \hat{f}_i(\alpha_{\mathbb{S}}\big(\bigcup_{i' \Rightarrow_\phi i} \phi(i')\big) & \\
& = \hat{f}_i\big(\bigsqcup_{i' \Rightarrow_\phi i} \alpha_2(\phi)(i')\big) & (\alpha_{\mathbb{S}} \text{ is distrib. and def. of } \alpha_2) \\
& \sqsubseteq \hat{f}_i\big(\bigsqcup_{i' \hookrightarrow_{\alpha_2(\phi)} i} \alpha_2(\phi)(i')\big) & \text{((1) of Def. 3.6)} \\
& = (\hat{F} \circ \alpha_2)(\phi)(i). & \text{(def. of } \hat{F}) \quad \square
\end{array}
$$

LEMMA 3.8 (SOUNDNESS). $\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$ *where* $\alpha = \alpha_2 \circ \alpha_1$.

PROOF. By Lemma 3.3, Lemma 3.7, and monotonicity of α_2. $\square$

3.5. Toward Sparse Analysis

The abstract semantic function given in (1) may propagate some abstract values unnecessarily between partitioning indices. For example, suppose that we analyze the statement $x := y$. We know for sure that the abstract semantic function for the statement *defines* a new abstract value only at variable x and *uses* only the abstract value of variable y. Thus, it is unnecessary to bring the whole of the abstract state to the statement; only the portions for variables x and y are enough. However, the function given in (1) blindly propagates the entire abstract states $\hat{\phi}(i')$ of all predecessors i' to the current partitioning index i.

To make the analysis sparse, we need to eliminate these unnecessary propagations by making the semantic function propagate abstract values only along the "semantic dependency," not blindly along the transition flows ($\hookrightarrow$)—that is, we make the semantic function propagate only the abstract values newly defined at one partitioning index to the other where they are actually used.

Our goal is to present a general framework for transforming static analysis defined as (1) into its sparse version while preserving the precision and soundness of the original analysis.

In the rest of this section, we will use the following example analysis to illustrate our framework. For simplicity, we use a simple analysis for a small imperative language, although our result is generally applicable to arbitrary static analyses and languages.

Example 3.9. We design a simple pointer analysis for a small subset of C. Suppose that a program is given as a control flow graph $\langle \mathbb{C}, \hookrightarrow \rangle$, where $\mathbb{C}$ is the set of control points and $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is the control flow relation among control points. Each control point is associated with a command. Consider the following simple subset of C:

$$\begin{array}{rcl} cmd & \rightarrow & x := e \mid *x := e \\ e & \rightarrow & x \mid \&x \mid *x. \end{array}$$

The meanings of each statement and each expression are fairly standard: a command is simply either a scalar assignment ($x := e$) or a pointer assignment ($*x := e$), which assigns the value of e into the variable x or the location to which x points, respectively. An expression may be a variable (x), an address-of expression ($\&x$), and a pointer dereference ($*x$).

We design a simple pointer analysis for this language. Suppose that we use the control-point partitioning (Section 3.4.1). Then, the abstract semantic function is defined as follows:

$$\hat{F}(\hat{\phi}) = \lambda i \in \mathbb{C}. \hat{f}_i\left(\bigsqcup_{i' \hookrightarrow i} \hat{\phi}(i')\right),$$

where the abstract domain is a map from control point to abstract states (i.e., $\hat{\phi} \in \mathbb{C} \rightarrow \hat{\mathbb{S}}$). In this example analysis, we suppose that control flow relation $\hookrightarrow$ is given before the analysis and hence does not depend on $\hat{\phi}$. The abstract state

$$\hat{s} \in \hat{\mathbb{S}} = Var \rightarrow \mathcal{P}(Var)$$

is a map from variables to their points-to variables. In this analysis, an abstract location is a program variable and an abstract value is a set of points-to variables.

The abstract semantic function $\hat{f}_i \in \hat{\mathbb{S}} \rightarrow \hat{\mathbb{S}}$ for commands is defined as follows ($\mathsf{cmd}(i)$ indicates the command associated with control point i):

$$\hat{f}_i(\hat{s}) = \begin{cases} \hat{s}[x \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(i) = x := e \\ \hat{s}[x' \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(i) = *x := e \text{ and } \hat{s}(x) = \{x'\} \\ \hat{s}[\hat{s}(x) \overset{w}{\mapsto} \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(i) = *x := e. \end{cases}$$

Auxiliary function $\hat{\mathcal{E}} \in e \rightarrow \hat{\mathbb{S}} \rightarrow \mathcal{P}(Var)$ evaluates the abstract value (points-to set) of e under the abstract state $\hat{s}$. The abstract effect of assignment $x := e$ is to replace the value of x by the value of e. With pointer assignment $*x := e$, we distinguish two cases: (1) when x points to a single location, we perform a strong update by replacing the value of the pointed location; (2) when x points to multiple locations, we make an weak update to the locations in $\hat{s}(x)$. The $\hat{\mathcal{E}}$ function is simply defined as follows:

$$\hat{\mathcal{E}}(e)(\hat{s}) = \begin{cases} \hat{s}(x) & e = x \\ \{x\} & e = \&x \\ \bigcup_{x' \in \hat{s}(x)} \hat{s}(x') & e = *x. \end{cases}$$

For variables (x), we look up the abstract state to find their abstract values. The abstract value of expression $\&x$ is $\{x\}$. The abstract value of expression $*x$ is obtained by joining all abstract values of variables in $\hat{s}(x)$.

3.6. Definition and Use Set

The first step toward deriving correct sparse analysis is to precisely define the notion of *definitions* and *uses*. Sparse analysis is derived based on these definitions and uses. Because we are interested in properties of the abstract semantics, they are defined in terms of the abstract semantic function $\hat{f}_i$.

Definition 3.10 (*Definition Set*). Definition set $\mathsf{D}(i)$ at partitioning index i is a set of abstract locations whose abstract values are ever changed by $\hat{f}_i$ during the analysis—that is (let $\mathcal{S} = \mathbf{lfp}\hat{F}$),

$$\mathsf{D}(i) = \left\{ l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i').\, \hat{f}_i(\hat{s})(l) \neq \hat{s}(l) \right\}.$$

In the definition, $\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i')$ denotes the input abstract state flowing to partitioning index i at the fixpoint, and therefore $\hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i')$ quantifies over the analysis process' intermediate states at partitioning index i. Thus, abstract location l is included in $\mathsf{D}(i)$ if and only if $\hat{f}_i$ changes the value of l at partitioning index i during the course of the analysis. In other words, if an abstract location l is not included in the definition set, the abstract semantic function has the identity transfer on l, which the following lemma states.

LEMMA 3.11. *For all $i \in \Delta, l \in \hat{\mathbb{L}}, \hat{s} \in \hat{\mathbb{S}}$,*

$$\left(l \notin \mathsf{D}(i) \wedge \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{(\mathbf{lfp}\hat{F})} i} (\mathbf{lfp}\hat{F})(i') \right) \implies \hat{f}_i(\hat{s})(l) = \hat{s}(l).$$

PROOF. Immediate from the dual statement of Definition 3.10. □

Note that the notion of definition set is a semantic one. For example, suppose that we analyze statement $x := x$. Even if the statement assigns a value to variable x, it has semantically no effect. Therefore, according to our definition, variable x is not included in the definition set of the statement.

Definition 3.12 (*Use Set*). Use set $\mathsf{U}(i)$ at partitioning index i consists of two parts:

$$\mathsf{U}(i) = \mathsf{U}_d(i) \cup \mathsf{U}_c(i).$$

The first part ($\mathsf{U}_d(i)$) is the set of abstract locations without which some values in $\mathsf{D}(i)$ are not properly generated—that is (let $\mathcal{S} = \mathbf{lfp}\hat{F}$),

$$\mathsf{U}_d(i) = \left\{ l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i').\, \hat{f}_i(\hat{s})|_{\mathsf{D}(i)} \neq \hat{f}_i(\hat{s}\backslash_l)|_{\mathsf{D}(i)} \right\}.$$

In addition, we collect abstract locations that are necessary to generate transition flows ($\hookrightarrow$): $\mathsf{U}_c(i)$ representing the set of abstract locations without which some flows in $\hookrightarrow_{\mathcal{S}}$ are not properly generated—that is,

$$\mathsf{U}_c(i) = \{ l \in \hat{\mathbb{L}} \mid \exists i' \in \Delta.(i, i') \in (\hookrightarrow_{\mathcal{S}}) \wedge (i, i') \notin (\hookrightarrow_{\mathcal{S}[i \mapsto \mathcal{S}(i)\backslash_l]}) \}.$$

Note that $\mathcal{S}[i \mapsto \mathcal{S}(i)\backslash_l]$ represents $\mathcal{S}$, whose abstract state in index i ($\mathcal{S}(i)$) does not contain an abstract value for location l ($\mathcal{S}(i)\backslash_l$). For static analysis of imperative languages, where transition relation $\hookrightarrow$ is given a priori and is not computed during the analysis, $\mathsf{U}_c(i)$ is $\emptyset$ and $\mathsf{U}(i)$ is identical to $\mathsf{U}_d(i)$.

Example 3.13. Suppose that we analyze the following program with the analysis designed in Example 3.9 (superscripts are control points):

$$^{⑩}x := \&y; \quad ^{⑪} * p := \&z; \quad ^{⑫}y := x; \tag{2}$$

Suppose further that the points-to set for pointer p at ⑪ is $\{x, y\}$ during the analysis. Then, according to the analysis definition in Example 3.9, abstract semantic function $\hat{f}_i$ for each control point i is as follows:

$$\begin{aligned}\hat{f}_{⑩}(\hat{s}) &= \hat{s}[x \mapsto \{y\}]\\ \hat{f}_{⑪}(\hat{s}) &= \hat{s}[\hat{s}(p) \overset{w}{\mapsto} \{z\}] = \hat{s}[\{x, y\} \overset{w}{\mapsto} \{z\}]\\ \hat{f}_{⑫}(\hat{s}) &= \hat{s}[y \mapsto \hat{s}(x)].\end{aligned}$$

Then, the definition set and use set at each control point are as follows:

$$\begin{aligned}\mathsf{D}(⑩) &= \{x\} & \mathsf{U}(⑩) &= \varnothing\\ \mathsf{D}(⑪) &= \{x, y\} & \mathsf{U}(⑪) &= \{p, x, y\}\\ \mathsf{D}(⑫) &= \{y\} & \mathsf{U}(⑫) &= \{x\}.\end{aligned}$$

The definition sets ($\mathsf{D}(i)$) are easy to check. Because $\hat{f}_{⑩}$ assigns a value to location x, $\mathsf{D}(⑩)$ includes x. Similarly, x and y are defined by $\hat{f}_{⑪}$, and y is defined by $\hat{f}_{⑫}$. For use sets ($\mathsf{U}(i)$), we compute $\mathsf{U}_d(i)$ only, since our example analysis does not update transition relation $\hookrightarrow$ during the analysis and hence $\mathsf{U}_c(i)$ is $\emptyset$. $\mathsf{U}(⑩)$ is $\emptyset$ because, according to the definition of $\hat{f}_{⑩}$, the values in $\mathsf{D}(⑩)(= \{x\})$ are generated without referring to any abstract location. $\mathsf{U}(⑪)$ includes p because p is dereferenced. In addition, $\mathsf{U}(⑪)$ includes x and y because of weak updates ($\overset{w}{\mapsto}$), $\hat{s}[\{x, y\} \overset{w}{\mapsto} \{z\}] = \hat{s}[x \mapsto \hat{s}(x) \cup \{z\}, y \mapsto \hat{s}(y) \cup \{z\}]$, where the values of x and y are referred. Note that this implicit use information, which does not explicitly appear in the program text, is naturally captured by following the abstract semantics. $\mathsf{U}(⑫)$ includes x whose value is referred to, to generate the value of y ($\mathsf{D}(⑫)$).

In the rest of the article, we frequently use a generalized notion of use set $\mathsf{U}_d(i)$.

Definition 3.14 (*Use Template*). We write $\mathbb{U}_Q(i)$ for the set of abstract locations that are necessary to properly generate the values in Q—that is (let $\mathcal{S} = \mathbf{lfp}\hat{F}$),

$$\mathbb{U}_Q(i) = \left\{ l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_S i} \mathcal{S}(i').\, \hat{f}_i(\hat{s})|_Q \neq \hat{f}_i(\hat{s}\backslash_l)|_Q \right\}.$$

Note that $\mathsf{U}_d(i) = \mathbb{U}_{\mathsf{D}(i)}(i)$.

Regarding the use set, we assume that abstract semantic function $\hat{f}_i$ and transition relation $\hookrightarrow$ are *well formed* in the following sense.

Definition 3.15 (*Well-Formed Abstract Semantic Function*). We say that abstract semantic function $\hat{f}_i$ is well formed if

$$\forall Q \subseteq \mathbb{L}, i \in \Delta, \hat{s} \in \hat{\mathbb{S}}, U \supseteq \mathbb{U}_Q(i).\, \hat{f}_i(\hat{s})|_Q = \hat{f}_i(\hat{s}|_U)|_Q.$$

The condition means that $\hat{f}_i$ properly generates the values of abstract locations in $\mathsf{D}(i)$ if the input state contains all of the use set, which naturally holds in most semantic functions in conventional static analysis. This requirement is not very important to understand the rest of the article but is necessary in the correctness proof (Appendix A). Similarly, we assume that $\hookrightarrow$ satisfies the following property.

Definition 3.16 (*Well-Formed Abstract Transition Relation*). We say that abstract transition relation $\hookrightarrow$ is well formed if

$$\forall i \in \Delta, U \supseteq \mathsf{U}_c(i), \hat{\phi} \in \Delta \to \hat{\mathbb{S}}.(\hookrightarrow_{\hat{\phi}}) = \left(\hookrightarrow_{\hat{\phi}[i \mapsto \hat{\phi}(i)|_U]} \right).$$

3.7. Data Dependencies

Once we have identified definition and use sets at each partitioning index, we can discover data dependencies of abstract semantic function $\hat{F}$ between two partitioning indices. Intuitively, if the abstract value of abstract location l defined at index i_0 is used at index i_n, there is a data dependency between i_0 and i_n on l. A formal definition of data dependency is given next.

Definition 3.17 (*Data Dependency*). Data dependency is quadruple relation $(\rightsquigarrow) \subseteq \Delta \times \hat{\mathbb{L}} \times \Delta \times (\Delta \to \hat{\mathbb{S}})$ defined as follows:

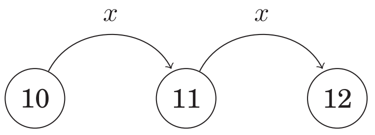
$$\begin{aligned} i_0 \overset{l}{\rightsquigarrow}_{\hat{\phi}} i_n \;\; \textit{iff} \;\; & \exists i_0 \ldots i_n \in \mathsf{Paths}(\hat{\phi}), l \in \hat{\mathbb{L}}. \\ & l \in \mathsf{D}(i_0) \cap \mathsf{U}(i_n) \wedge \forall k \in (0, n).l \notin \mathsf{D}(i_k), \end{aligned} \tag{3}$$

where $\mathsf{Paths}(\hat{\phi})$ is the set of all paths created by transition relation $\hookrightarrow_{\hat{\phi}}$: a path $p = p_0 p_1 \cdots p_n$ is a sequence of partitioning indices such that $p_0 \hookrightarrow_{\hat{\phi}} p_1 \hookrightarrow_{\hat{\phi}} \cdots \hookrightarrow_{\hat{\phi}} p_n$, then,

$$\mathsf{Paths}(\hat{\phi}) = \mathbf{lfp}\lambda P.\{i_0 i_1 \mid i_0 \hookrightarrow_{\hat{\phi}} i_1\} \cup \{p_0 p_1 \cdots p_n i \mid p \in P \wedge p_n \hookrightarrow_{\hat{\phi}} i\}.$$

The data dependency $i_0 \overset{l}{\rightsquigarrow}_{\hat{\phi}} i_n$ means that if there exists a path from partitioning index i_0 to i_n, a value of abstract location l can be defined at i_0 and used at i_n, and there is no intermediate indices i_k that may change the value of l, then a data dependency exists between partitioning indices i_0 and i_n on location l.

Example 3.18. In the program presented in Example 3.13, we can find two data dependencies, ⑩ $\overset{x}{\rightsquigarrow}$ ⑪ and ⑪ $\overset{x}{\rightsquigarrow}$ ⑫, as graphically depicted as follows:



We omit the subscript $\hat{\phi}$ from $\rightsquigarrow$ when the transition relation is determined without $\hat{\phi}$.

3.8. Sparse Abstract Semantics Function

Using data dependency, we can make abstract semantic function sparse by propagating between partitioning indices only the abstract values that participate in the fixpoint computation. Sparse abstract semantic function $\hat{F}_s$, whose definition follows, is the same as the original one (1) except that it propagates abstract values along the data dependency, not along the transition relation:

$$\hat{F}_s(\hat{\phi}) = \lambda i \in \Delta. \hat{f}_i \left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_l \right). \tag{4}$$

Compared to the dense abstract semantic function (1), this definition is different only in that it is defined over data dependency $(\rightsquigarrow)$, so we can reuse semantic function $\hat{f}_i$ and its soundness result $\alpha_{\mathbb{S}} \circ f_i \sqsubseteq \hat{f}_i \circ \alpha_{\mathbb{S}}$ from the original analysis design.

The following theorem states that the analysis result with the sparse abstract semantic function is the same as the one of original analysis.

THEOREM 3.19 (CORRECTNESS).

$$\forall i \in \Delta.\forall l \in \mathsf{D}(i).(\mathbf{lfp}\hat{F}_s)(i)(l) = (\mathbf{lfp}\hat{F})(i)(l).$$

PROOF. Shortly, we will notice that this theorem is a corollary of Theorem 3.23, where $\hat{F}_s$ is an instance of $\hat{F}_a$ such that $\hat{\mathsf{D}}(i) = \mathsf{D}(i)$ and $\hat{\mathsf{U}}(i) = \mathsf{U}(i)$. □

The theorem guarantees that the sparse analysis result is identical to the original result only up to the entries that are defined at every partitioning index. Note that we can also show the complete equivalence between $\mathbf{lfp}\hat{F}$ and $\mathbf{lfp}\hat{F}_s$ by reconstructing the missing entries (see Appendix B).

3.9. Sparse Analysis with Approximated Data Dependency

The sparse analysis designed until Section 3.8 is not practical. The definitions of D and U are purely mathematical but nonconstructive, and they are defined in terms of the original fixpoint $\mathbf{lfp}\hat{F}$.

We now design a practical sparse analysis. The practicality is obtained by approximating the definition and the use sets. Note that the initial precision and soundness of the original analysis are still preserved even with the approximations if some safety conditions are satisfied. We discuss the safety conditions in Section 3.9.1. Suppose that $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are such safe approximations of D and U, respectively. With $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$, we can approximate the data dependency.

Definition 3.20 (*Approximated Data Dependency*). Approximated data dependency is quadruple relation $(\rightsquigarrow) \subseteq \Delta \times \hat{\mathbb{L}} \times \Delta \times (\Delta \to \hat{\mathbb{S}})$ defined as follows:

$$\begin{aligned} i_0 \overset{l}{\rightsquigarrow}_{\hat{\phi}} i_n \ \ \textit{iff} \ \ & \exists i_0 \ldots i_n \in \mathsf{Paths}(\hat{\phi}), l \in \hat{\mathbb{L}}. \\ & l \in \hat{\mathsf{D}}(i_0) \cap \hat{\mathsf{U}}(i_n) \wedge \forall k \in (0, n).l \notin \hat{\mathsf{D}}(i_k). \end{aligned}$$

The definition is the same as (3) except that it is defined over $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. The derived sparse analysis is to compute the fixpoint of the following abstract semantic function:

$$\hat{F}_a(\hat{\phi}) = \lambda i \in \Delta.\, \hat{f}_i \left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_l \right). \tag{5}$$

$\hat{F}_a$ is the same as $\hat{F}_s$ except that $\hat{F}_a$ is defined over the approximated data dependency.

3.9.1. Conditions for Safe Approximations. For the approximation to be safe (i.e., still $\mathbf{lfp}\hat{F} = \mathbf{lfp}\hat{F}_a$), $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ should satisfy two conditions.

(1) Both $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ are overapproximations of $\mathsf{D}(i)$ and $\mathsf{U}(i)$, respectively.
(2) Abstract locations that are necessary to generate values of spurious definitions $(\hat{\mathsf{D}}(i) - \mathsf{D}(i))$ should be also included in $\hat{\mathsf{U}}(i)$.

The first condition is intuitive, and we can easily show that the analysis computes different results if one of them is not an overapproximation. Regarding the second condition, the following example illustrates what happens when there exists an abstract location that is used to generate spurious definitions but is not included in the approximated use set.

Formally, safe approximations of definition and use sets are as follows.

Definition 3.21 (*Safe Approximations of* D *and* U). We say that $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations of D and U, respectively, if and only if

(1) $\hat{\mathsf{D}}(i) \supseteq \mathsf{D}(i) \wedge \hat{\mathsf{U}}(i) \supseteq \mathsf{U}(i)$
(2) $\hat{\mathsf{U}}(i) \supseteq \Cup_{(\hat{\mathsf{D}}(i) \setminus \mathsf{D}(i))}(i)$.

Example 3.22. Suppose that we analyze the following program with the original analysis designed in Example 3.9:

$$^{⑩}x := \&y; \quad ^{⑪}* p := \&z; \quad ^{⑫}y := x;$$

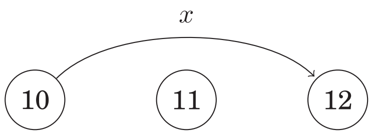
Suppose further that the points-to set for pointer p at ⑪ is $\{y\}$ during the original analysis. Then, according to the analysis definition in Example 3.9, abstract semantic function $\hat{f}_i$ for each control point i is as follows:

$$\begin{aligned} \hat{f}_{⑩}(\hat{s}) &= \hat{s}[x \mapsto \{y\}] \\ \hat{f}_{⑪}(\hat{s}) &= \hat{s}[y \mapsto \{z\}] \\ \hat{f}_{⑫}(\hat{s}) &= \hat{s}[y \mapsto \hat{s}(x)]. \end{aligned}$$

Then, definition sets and use sets are as follows:

$$\begin{aligned} \mathsf{D}(⑩) &= \{x\} & \mathsf{U}(⑩) &= \varnothing \\ \mathsf{D}(⑪) &= \{y\} & \mathsf{U}(⑪) &= \{p\} \\ \mathsf{D}(⑫) &= \{y\} & \mathsf{U}(⑫) &= \{x\}. \end{aligned}$$

With these definition and use sets, one data dependency ⑩ $\overset{x}{\rightsquigarrow}$ ⑫ is generated as follows:

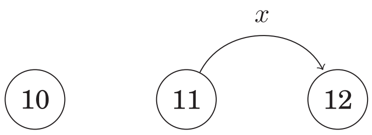


For a sparse version of the original analysis, we need to approximate the definition and use sets. However, note that not all overapproximations make the sparse analysis safe.

The following is one example of overapproximations of the preceding D and U, yet they are unsafe:

$$\begin{aligned} \hat{\mathsf{D}}(⑩) &= \{x\} & \hat{\mathsf{U}}(⑩) &= \varnothing \\ \hat{\mathsf{D}}(⑪) &= \{x, y\} & \hat{\mathsf{U}}(⑪) &= \{p\} \\ \hat{\mathsf{D}}(⑫) &= \{y\} & \hat{\mathsf{U}}(⑫) &= \{x\}. \end{aligned}$$

Here, x's definition at ⑪ is spurious because of the approximation. With this approximation, we generate one data dependency ⑪ $\overset{x}{\rightsquigarrow}$ ⑫:



Because of the spurious definition of x at ⑪, the x's definition at ⑩ does not reach to ⑫, which makes the subsequent main analysis unsafe.

To fix this problem, we need the second condition of the safe approximations ((2) of Definition 3.21): we adjust $\hat{\mathsf{U}}$ to include locations that are necessary to generate values of spurious definitions. For our example, we make $\hat{\mathsf{U}}(⑪)$ include the spurious definition of x:

$$\begin{aligned} \hat{\mathsf{D}}(⑩) &= \{x\} & \hat{\mathsf{U}}(⑩) &= \varnothing \\ \hat{\mathsf{D}}(⑪) &= \{x, y\} & \hat{\mathsf{U}}(⑪) &= \{p, x\} \\ \hat{\mathsf{D}}(⑫) &= \{y\} & \hat{\mathsf{U}}(⑫) &= \{x\}. \end{aligned}$$

With this approximation, we generate two data dependencies, $⑩ \overset{x}{\rightsquigarrow} ⑪$ and $⑪ \overset{x}{\rightsquigarrow} ⑫$:

$$\begin{tikzcd} ⑩ \arrow[r, bend left, "x"] & ⑪ \arrow[r, bend left, "x"] & ⑫ \end{tikzcd}$$

Following these two data dependencies, the abstract value of x at ⑩ will be propagated to ⑫ in the subsequent main analysis. Note that in the main analysis, x is not modified at ⑪: the approximated definitions ($\hat{\mathsf{D}}$) and uses ($\hat{\mathsf{U}}$) are used only for the generation of data dependencies. The main analysis (fixpoint computation) is performed following these preconstructed paths with the original abstract semantic function $\hat{f}_i$ that does not involve spurious definitions. This is why our sparse analysis with approximated def-use paths does not degrade the analysis precision.

Formally, we can prove that the safe approximations $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ yield the correct sparse analysis, which the following theorem states:

THEOREM 3.23 (CORRECTNESS). *Suppose that sparse abstract semantic function $\hat{F}_a$ is derived by safe approximations $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. Then,*

$$\forall i \in \Delta. \forall l \in \hat{\mathsf{D}}(i).(\mathbf{lfp}\hat{F}_a)(i)(l) = (\mathbf{lfp}\hat{F})(i)(l).$$

PROOF. See Appendix A. □

3.10. Precision Loss with Conventional Def-Use Chains

Our notion of data dependency is different from the conventional notion of def-use chains. Conventional def-use chains connect each definition to every possible use of the definition. We can express this def-use chain relation $\frown$ as follows:

Definition 3.24 (*Def-Use Chains*).

$$\begin{aligned} i_0 \overset{l}{\frown}_{\hat{\phi}} i_n \ \textit{iff} \ & \exists i_0 \dots i_n \in \mathsf{Paths}(\hat{\phi}), l \in \hat{\mathbb{L}}. \\ & l \in \hat{\mathsf{D}}(i_0) \cap \hat{\mathsf{U}}(i_n) \wedge \forall k \in (0, n).l \notin \hat{\mathsf{D}}_{\mathsf{must}}(i_k), \end{aligned}$$

where $\hat{\mathsf{D}}_{\mathsf{must}}(i)$ denotes the set of abstract locations that are "must' defined (killed) at i.

The only difference from ours is the use of $\hat{\mathsf{D}}_{\mathsf{must}}$ in place of $\hat{\mathsf{D}}$. Although our definition of data dependency does not degrade the precision of the resulting sparse analysis, this conventional def-use chains would have made the analysis less precise. The following example illustrates the case of imprecision.

Example 3.25. Suppose that we analyze the following program with the original analysis designed in Example 3.9:

$$^{⑩}x := \&y; \quad ^{⑪}*p := \&z; \quad ^{⑫}y := x;$$

Suppose further that the points-to set for pointer p at ⑪ is $\{x\}$ during the original analysis. Note that because of the strong update to x at ⑪, the value of x is $\{z\}$ at ⑫ during the original analysis. Suppose, for sparse analysis, that we use the following approximated definition and use sets during the def-use path construction:

$$\begin{array}{llll}
\hat{\mathsf{D}}(\text{⑩}) = \{x\} & \hat{\mathsf{D}}_{\mathsf{must}}(\text{⑩}) = \{x\} & \hat{\mathsf{U}}(\text{⑩}) = \varnothing \\
\hat{\mathsf{D}}(\text{⑪}) = \{x, y\} & \hat{\mathsf{D}}_{\mathsf{must}}(\text{⑪}) = \varnothing & \hat{\mathsf{U}}(\text{⑪}) = \{p, x, y\} \\
\hat{\mathsf{D}}(\text{⑫}) = \{y\} & \hat{\mathsf{D}}_{\mathsf{must}}(\text{⑫}) = \{y\} & \hat{\mathsf{U}}(\text{⑫}) = \{x\}.
\end{array}$$

With this information, the conventional def-use chains (Definition 3.24) become as follows:

$$\begin{tikzcd}
\boxed{10} \arrow[r, "x"'] \arrow[rr, bend left, "x"] & \boxed{11} \arrow[r, "x"'] & \boxed{12}
\end{tikzcd}$$

Please note that the ⑫ point becomes a join point that degrades the precision of the subsequent, main analysis: the value of x at ⑫ is $\{y\} \cup \{z\}$ that is bigger than $\{z\}$, the one that appears in the original analysis.

Meanwhile, our data dependency (Definition 3.20) builds the def-use paths as follows:

$$\begin{tikzcd}
\boxed{10} \arrow[r, bend left, "x"] & \boxed{11} \arrow[r, bend left, "x"] & \boxed{12}
\end{tikzcd}$$

Note that there are no join points in the preceding data dependencies: the main analysis along these data dependencies does not degrade the analysis precision at ⑫.

3.11. Existing Sparse Analyses as Instances

In this subsection, we show that recent two sparse analysis techniques [Hardekopf and Lin 2009, 2011] can be understood as specific instances of our framework that approximate the definition and use sets in certain ways.

Nonsparse analysis. Before discussing those two existing sparse techniques, we first show a simple example that nonsparse analysis (baseline analysis given in (1)) can be also understood as a sparse analysis that approximates definition and use sets in a very crude way:

$$\begin{aligned}
\hat{\mathsf{D}}(i) &= \hat{\mathbb{L}} \\
\hat{\mathsf{U}}(i) &= \hat{\mathbb{L}}.
\end{aligned}$$

That is, we approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ in a way that all abstract locations are considered as definitions and uses at every partitioning index. Then, the sparse analysis with such $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ is identical to the conventional nonsparse analysis, which brings the full abstract state (the abstract values of all locations) along all partitioning indices.

Semisparse analysis. The challenge of sparse analysis in general is that the definition and use sets are not available prior to the analysis. The semisparse technique [Hardekopf and Lin 2009] solves this problem by exploiting the fact that def-use information for top-level variables (variables that are not address taken in the program) are available before the analysis. Thus, the technique performs sparse analysis on such variables while using the conventional nonsparse analysis on the other variables. This technique is conceptually identical to our sparse analysis with the following $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$:

$$\begin{aligned}\hat{\mathsf{D}}(i) &= \{l \in \hat{\mathbb{L}} \mid \hat{s} \in \hat{\mathbb{S}} \wedge \hat{f}_i(\hat{s})(l) \neq \hat{s}(l)\}\\ \hat{\mathsf{U}}(i) &= \mathbb{U}_{\hat{\mathsf{D}}(i)}(i).\end{aligned}$$

The $\hat{\mathsf{D}}(i)$ includes an abstract location if semantic function $\hat{f}_i$ can change the value of the location for an arbitrary input state $\hat{s} \in \hat{\mathbb{S}}$. For example, for statement $x := y$, $\hat{\mathsf{D}}$ includes only x and $\hat{\mathsf{U}}$ includes only y, but for statement $*p := 1$, $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ include all abstract locations because p may point to arbitrary locations under arbitrary input states. Thus, in sparse analysis with such $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$, some statements such as $x := y$ are analyzed sparsely, but $*p := 1$ is analyzed densely.

Staged-sparse analysis. The staged-sparse analysis [Hardekopf and Lin 2011] solves the challenge of sparse analysis by employing a preanalysis and computing conservative def-use information. This idea is formalized in our framework as follows. First, we compute an overapproximation $\hat{X}$ of $\mathbf{lfp}\hat{F}$ (i.e., $\hat{X} \sqsupseteq \mathbf{lfp}\hat{F}$) by a preanalysis. Next, we approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ using $\hat{X}$:

$$\begin{aligned}\hat{\mathsf{D}}(i) &= \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \textstyle\bigsqcup_{i' \hookrightarrow i} \hat{X}(x').\, \hat{f}_i(\hat{s})(l) \neq \hat{s}(l)\}\\ \hat{\mathsf{U}}(i) &= \mathbb{U}_{\hat{\mathsf{D}}(i)}(i).\end{aligned}$$

This approximation method is more accurate than that of semisparse analysis. For example, in statement $*p := 1$, the semisparse technique considers all abstract locations as its definitions. On the other hand, staged-sparse technique considers definitions only up to $\hat{X}$. Suppose that p may point to $\{x, y\}$ in $\hat{X}$. Then, only x and y are considered as definitions of the statement.

3.12. Designing Sparse Analysis Steps in the Framework

In summary, the design of sparse analysis within our framework is done in the following two steps:

(1) Design a nonsparse static analysis based on the abstract interpretation framework. Note that the abstract domain should be a member of the family explained in Section 3.4.
(2) Design a method to find a safe approximation $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ of definition set D and use set U (Definition 3.21).

In practice, the second step (finding $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$) is the challenging part. In the following sections, we show that for C-like languages, $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ can be effectively computed by using a flow-insensitive version of the main analysis. However, it remains to be seen how to design an effective such preanalysis for other languages.

4. DESIGNING SPARSE NONRELATIONAL ANALYSIS

In this section, we present an example of designing sparse analysis for nonrelational numeric analyses for C-like imperative languages. Following Section 3.12, we first define a nonsparse analysis and then show how to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ that satisfy the safe approximation conditions (Definition 3.21). The sparse analysis designed in this section is the core of our interval domain-based static analyzer, called Interval$_{\text{sparse}}$, which will be evaluated in Section 7.

4.1. Step 1: Designing Nonsparse Analysis

Language. For brevity, we restrict our presentation to the following simple subset of C, where a variable has either an integer value or a pointer:

$$\begin{array}{c} cmd \to x := e \mid *x := e \mid \{\!\{x < n\}\!\} \\ \text{where } e \to n \mid x \mid \&x \mid *x \mid e{+}e. \end{array}$$

Assignment $x := e$ corresponds to assigning the value of expression e to variable x. Store $*x := e$ performs indirect assignments; the value of e is assigned to the location to which x points. An assume command $\{\!\{x < n\}\!\}$ makes the program continue only when the condition evaluates to true.

Abstract domain. We consider an analysis that overapproximates the reachable states for each control point: the abstract domain is a map from $\mathbb{C} \to \hat{\mathbb{S}}$, where $\mathbb{C}$ is the set of control points in the program and $\hat{\mathbb{S}}$ is a nonrelational abstract state such that $\mathcal{P}(\mathbb{S}^+) \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$:

$$\begin{aligned} \hat{\mathbb{S}} &= \hat{\mathbb{L}} \to \hat{\mathbb{V}} \\ \hat{\mathbb{L}} &= Var \\ \hat{\mathbb{V}} &= \hat{\mathbb{Z}} \times \hat{\mathbb{P}} \\ \hat{\mathbb{P}} &= \mathcal{P}(\hat{\mathbb{L}}). \end{aligned}$$

Abstract state $\hat{\mathbb{S}}$ is a map from abstract locations $\hat{\mathbb{L}}$ to abstract values $\hat{\mathbb{V}}$. An abstract location is a program variable. An abstract value is a pair of an abstract integer $\hat{\mathbb{Z}}$ and an abstract pointer $\hat{\mathbb{P}}$. A set of integers is abstracted into an abstract integer ($\mathcal{P}(\mathbb{Z}) \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \hat{\mathbb{Z}}$). Note that the abstraction is generic, so we can choose any nonrelational numeric domains of our interest, such as intervals ($\hat{\mathbb{Z}} = \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \leq u\} \cup \{\bot\}$). For simplicity, we do not abstract pointers (because they are finite): pointer values are kept by a points-to set ($\hat{\mathbb{P}} = \mathcal{P}(\hat{\mathbb{L}})$). Other pointer abstractions are also orthogonally applicable.

Abstract semantics. The abstract semantics is defined by the least fixpoint of the following semantic function:

$$\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$$

$$\hat{F}(\hat{\phi}) = \lambda i \in \mathbb{C}.\, \hat{f}_i\left(\bigsqcup_{i' \hookrightarrow i} \hat{\phi}(i')\right).$$

Note that we suppose the control flows of the program is known and $\hookrightarrow$ does not depend on analysis states ($\hat{\phi}$). We define the semantic function $\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ as follows ($\mathsf{cmd}(i)$ denotes the command associated with control point i):

$$\hat{f}_i(\hat{s}) = \begin{cases} \hat{s}[x \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(i) = x := e \\ \hat{s}[\hat{s}(x).\hat{\mathbb{P}} \overset{w}{\mapsto} \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(i) = *x := e \\ \hat{s}[x \mapsto \langle \hat{s}(x).\hat{\mathbb{Z}} \sqcap_{\hat{\mathbb{Z}}} \alpha_{\mathbb{Z}}(\{z \in \mathbb{Z} | z < n\}), \hat{s}(x).\hat{\mathbb{P}}\rangle] & \mathsf{cmd}(i) = \{\{x < n\}\}. \end{cases}$$

Auxiliary function $\hat{\mathcal{E}}(e)(\hat{s})$ computes abstract value of e under $\hat{s}$. Assignment $x := e$ updates the value of x. Store $*x := e$ weakly updates the value of abstract locations that $*x$ denotes.[2] $\{\{x < n\}\}$ confines the interval value of x according to the condition.

[2]For brevity, we consider only weak updates. Applying a strong update is orthogonal to our sparse analysis design.

$\hat{\mathcal{E}} \in e \to \hat{\mathbb{S}} \to \hat{\mathbb{V}}$ is defined as follows:

$$\begin{array}{rcl}
\hat{\mathcal{E}}(n)(\hat{s}) & = & \langle \alpha_{\mathbb{Z}}(\{n\}), \bot \rangle \\
\hat{\mathcal{E}}(x)(\hat{s}) & = & \hat{s}(x) \\
\hat{\mathcal{E}}(\&x)(\hat{s}) & = & \langle \bot, \{x\} \rangle \\
\hat{\mathcal{E}}(*x)(\hat{s}) & = & \bigsqcup\{\hat{s}(a) \mid a \in \hat{s}(x).\hat{\mathbb{P}}\} \\
\hat{\mathcal{E}}(e_1 + e_2)(\hat{s}) & = & \langle v_1.\hat{\mathbb{Z}} \hat{+}_{\hat{\mathbb{Z}}} v_2.\hat{\mathbb{Z}}, v_1.\hat{\mathbb{P}} \cup v_2.\hat{\mathbb{P}} \rangle \\
 & & \text{where } v_1 = \hat{\mathcal{E}}(e_1)(\hat{s}),\ v_2 = \hat{\mathcal{E}}(e_2)(\hat{s}).
\end{array}$$

Note that the preceding analysis is parameterized by an abstract numeric domain $\hat{\mathbb{Z}}$ and sound operators $\hat{+}_{\hat{\mathbb{Z}}}$ and $\sqcap_{\hat{\mathbb{Z}}}$. In this section, we assume that $\hookrightarrow$ is fixed and given prior to the analysis, which is an acceptable assumption for C-like languages.

4.2. Step 2: Finding Definitions and Uses

The second step is to find safe approximations of definitions and uses. The sparse analysis framework provides a mathematical definitions regarding correctness but does not provide how to find safe $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. In the rest of this section, we present a semantics-based systematic way to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$.

We propose to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ from a conservative approximation of $\hat{F}$. We call the approximated analysis by preanalysis. Let $\hat{\mathbb{D}}_{pre}$ and $\hat{F}_{pre}$ be the domain and semantic function of such a preanalysis, which satisfies the following two conditions:

$$\begin{array}{c}
\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha_{pre}}{\overset{\gamma_{pre}}{\leftrightarrows}} \hat{\mathbb{D}}_{pre} \\
\alpha_{pre} \circ \hat{F} \sqsubseteq \hat{F}_{pre} \circ \alpha_{pre}
\end{array}.$$

By the abstract interpretation framework [Cousot and Cousot 1977, 1979, 1992], such a preanalysis is guaranteed to be conservative (i.e., $\alpha_{pre}(\mathbf{lfp}\hat{F}) \sqsubseteq \mathbf{lfp}\hat{F}_{pre}$). As an example, in our experiments (Section 7), we use a simple abstraction as follows:

$$\alpha_{pre} = \lambda \hat{X}. \bigsqcup\{\hat{X}(i) \mid i \in \mathsf{dom}(\hat{X})\}.$$

The abstract semantic function is defined as

$$\hat{F}_{pre} = \lambda \hat{s}. \bigsqcup_{i \in \mathbb{C}} \hat{f}_i(\hat{s}).$$

The abstraction ignores the control flows of programs and computes a single global invariant (a.k.a., flow insensitivity).

We now define $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ using the preanalysis result. Let $\hat{s}_{pre} \in \hat{\mathbb{S}}$ be the preanalysis result. The definitions of $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are derived from the semantic definition of $\hat{f}_i$:

$$\hat{\mathsf{D}}(i) = \begin{cases} \{x\} & \mathsf{cmd}(i) = x := e \\ \hat{s}_{pre}(x).\hat{\mathbb{P}} & \mathsf{cmd}(i) = *x := e \\ \{x\} & \mathsf{cmd}(i) = \{\!\{x < n\}\!\}. \end{cases}$$

$\hat{\mathsf{D}}$ includes locations whose values are potentially defined (changed). In the definition of $\hat{f}_i$ for $x := e$ and $\{\!\{x < n\}\!\}$, we notice that abstract location x may be defined. In $*x := e$, we see that $\hat{f}_i$ may define locations $\hat{s}(x).\hat{\mathbb{P}}$ for a given input state $\hat{s}$ at program point c. Here, we use the preanalysis: because we cannot have the input state $\hat{s}$ prior to the analysis, we instead use its conservative abstraction $\hat{s}_{pre}$. Such $\hat{\mathsf{D}}$ satisfies the safe approximation condition (Definition 3.21) because we collect all potentially defined locations, preanalysis is conservative, and $\hat{f}_i$ is monotone.

Before defining $\hat{\mathsf{U}}$, we define an auxiliary function $\mathcal{U} \in e \to \hat{\mathbb{S}} \to \mathcal{P}(\hat{\mathbb{L}})$. Given expression e and state $\hat{s}$, $\mathcal{U}(e)(\hat{s})$ finds the set of abstract locations that are referenced during the evaluation of $\hat{\mathcal{E}}(e)(\hat{s})$. Thus, $\mathcal{U}$ is naturally derived from the definition of $\hat{\mathcal{E}}$:

$$\begin{aligned}
\mathcal{U}(n)(\hat{s}) &= \emptyset \\
\mathcal{U}(x)(\hat{s}) &= \{x\} \\
\mathcal{U}(\&x)(\hat{s}) &= \emptyset \\
\mathcal{U}(*x)(\hat{s}) &= \{x\} \cup \hat{s}(x).\hat{\mathbb{P}} \\
\mathcal{U}(e_1{+}e_2)(\hat{s}) &= \mathcal{U}(e_1)(\hat{s}) \cup \mathcal{U}(e_2)(\hat{s}).
\end{aligned}$$

When e is either n or $\&x$, $\hat{\mathcal{E}}$ does not refer any abstract location. Because $\hat{\mathcal{E}}(x)(\hat{s})$ references abstract location x, $\mathcal{U}(x)(\hat{s})$ is defined by $\{x\}$. $\hat{\mathcal{E}}(*x)(\hat{s})$ references location x and each location $a \in \hat{s}(x)$, thus the set of referenced locations is $\{x\} \cup \hat{s}(x).\hat{\mathbb{P}}$. $\hat{\mathsf{U}}$ is defined as follows:

$$\hat{\mathsf{U}}(i) = \begin{cases} \mathcal{U}(e)(\hat{s}_{pre}) & \mathsf{cmd}(i) = x := e \\ \{x\} \cup \hat{s}_{pre}(x).\hat{\mathbb{P}} \cup \mathcal{U}(e)(\hat{s}_{pre}) & \mathsf{cmd}(i) = *x := e \\ \{x\} & \mathsf{cmd}(i) = \{\!\{x < n\}\!\}. \end{cases}$$

Using $\hat{s}_{pre}$ and $\mathcal{U}$, we collect abstract locations that are potentially used during the evaluation of e. Because $\hat{f}_i$ is defined to refer to abstract location x in $*x := e$ and $\{\!\{x < n\}\!\}$, $\mathcal{U}$ additionally includes x. Note that in $*x := e$, $\hat{\mathsf{U}}(c)$ includes $\hat{s}_{pre}(x).\hat{\mathbb{P}}$ because the abstract semantics ($\hat{f}_i$'s definition) performs weak updates. Because we define $\hat{\mathsf{U}}(i)$ in a way that it includes the entire $\hat{\mathsf{D}}(i)$, it is easy to verify that $\hat{\mathsf{U}}(i)$ satisfies the conditions in Definition 3.21.

LEMMA 4.1. *$\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations.*

5. DESIGNING SPARSE RELATIONAL ANALYSIS

In this section, we design a sparse relational analysis. We define a family of relational analyses (Section 5.1) and show a safe approximation of definitions $\hat{\mathsf{D}}$ and uses $\hat{\mathsf{U}}$ for the analysis (Section 5.2).

We consider *packed* relational analysis [Blanchet et al. 2003; Miné 2006b]. A pack is a semantically related set of variables. We assume that a set of variable packs, $\mathit{Packs} \subseteq \mathcal{P}(\mathit{Var})$ such that $\bigcup \mathit{Packs} = \mathit{Var}$ (the packs can overlap), are given by users or a preanalysis [Miné 2006b; Cousot et al. 2009]. In a packed relational analysis, abstract states ($\hat{\mathbb{S}}$) map variable packs (Packs) to a relational domain ($\hat{\mathbb{R}}$) (i.e., $\hat{\mathbb{S}} = \mathit{Packs} \to \hat{\mathbb{R}}$).

The distinguishing feature of sparse relational analysis is that definition sets and use sets are defined in terms of variable packs. For example, in a simple statement $x := 1$, all variable packs that contain x may be defined and used at the same time, whereas only variable x may be defined and not used in nonrelational analysis. As a result, data dependencies are also defined in terms of variable packs. We denote a pack of variables $x_1, \ldots, x_n$ as $\langle\!\langle x_1, \ldots, x_n \rangle\!\rangle$.

5.1. Step 1: Designing Nonsparse Analysis

We consider a packed relational analysis based on the octagon abstract domain [Miné 2006b]. This is for the clarity of the presentation, and the overall idea is applicable to other relational domains such as the polyhedron [Cousot and Halbwachs 1978].

Language. We consider commands where numeric constants $c \in \mathbb{Z}$ are enhanced into constant intervals $[a, b]$, where $a \in \mathbb{Z} \cup \{-\infty\}$ and $b \in \mathbb{Z} \cup \{+\infty\}$. This allows not only modeling nondeterministic behaviors of programs, such as user inputs, but also simplifying the abstract semantics of relational analysis with variable packs. Formally, we consider the following commands:

$$cmd \;\rightarrow\; x := [a, b] \mid x := \pm y + [a, b].$$

Note that these two types of assignments are the ones that the octagon domain is able to precisely handle. Other assignment forms can be handled approximately via conversions to interval or polyhedron domains [Miné 2006b]. We do not consider pointers: including pointers in the language does not require novelty but verbosity. We focus only on the differences between nonrelational and relational sparse analysis designs.

Abstract domain. We consider an analysis that computes an overapproximation of reachable states for each control point, so the abstract domain is a map from control points to abstract states (i.e., $\mathbb{C} \rightarrow \hat{\mathbb{S}}$), where $\mathbb{C}$ is the set of control points in the program. In packed relational analyses, the abstract state ($\hat{\mathbb{S}}$) is a map from variable packs to relational domain elements:

$$\hat{\mathbb{S}} = \mathit{Packs} \rightarrow \hat{\mathbb{R}}.$$

We suppose that the following operators are given:

—$\hat{R} \in \mathit{cmd} \rightarrow \hat{\mathbb{R}} \rightarrow \hat{\mathbb{R}}$: A semantic function of the relational domain for commands. For the octagon domain, the definitions of these operators are available in Miné [2006b].
—$\mathsf{toInt}_x \in \hat{\mathbb{S}} \rightarrow \hat{\mathbb{I}}$: A projection function that projects a relational domain element onto variable x to obtain x's interval value ($\hat{\mathbb{I}}$: the lattice of intervals). To be safe, toInt_x should satisfy the following condition:

$$\forall \hat{s} \in \hat{\mathbb{S}}.\mathsf{toInt}_x(\hat{s}) \sqsupseteq \alpha_{\hat{\mathbb{I}}}\left(\bigcap\nolimits_{p \in \mathsf{pack}(x)} \{s(x) \mid s \in \gamma_{\hat{\mathbb{R}}}(\hat{s}(p))\}\right),$$

where $\alpha_{\hat{\mathbb{I}}}$ is the abstraction function for the lattice of intervals such that $\mathcal{P}(\mathbb{Z}) \underset{\alpha_{\hat{\mathbb{I}}}}{\overset{\gamma_{\hat{\mathbb{I}}}}{\leftrightarrows}} \hat{\mathbb{I}}$, $\alpha_{\hat{\mathbb{R}}}$ is the abstraction function for the octagon domain such that $\mathcal{P}(\mathbb{S}) \underset{\alpha_{\hat{\mathbb{R}}}}{\overset{\gamma_{\hat{\mathbb{R}}}}{\leftrightarrows}} \hat{\mathbb{R}}$, and $\mathsf{pack}(x)$ is the set of packs that contain x (i.e., $\mathsf{pack}(x) = \{p \in \text{Packs} \mid x \in p\}$).

Abstract semantics. The abstract semantics is defined by the least fixpoint of the following function:

$$\hat{F} \in (\mathbb{C} \rightarrow \hat{\mathbb{S}}) \rightarrow (\mathbb{C} \rightarrow \hat{\mathbb{S}})$$
$$\hat{F}(\hat{\phi}) = \lambda i \in \mathbb{C}.\hat{f}_i\left(\bigsqcup_{i' \hookrightarrow i} \hat{\phi}(i')\right).$$

The abstract semantic function $\hat{f}_i$ for the octagon domain is defined as follows:

$$\hat{f}_i(\hat{s}) \;=\; \lambda p \in \mathit{Packs}.\begin{cases} \hat{R}(x := [a, b])(\hat{s}(p)) & \mathsf{cmd}(c) = x := [a, b] \wedge x \in p \\ \hat{R}(x := \pm y + [a, b])(\hat{s}(p)) & \mathsf{cmd}(c) = x := \pm y + [a, b] \wedge x \in p \wedge y \in p \\ \hat{R}(x := \pm \mathsf{toInt}_y(\hat{s}) + [a, b])(\hat{s}(p)) & \mathsf{cmd}(c) = x := \pm y + [a, b] \wedge x \in p \wedge y \notin p \\ \hat{s}(p) & \text{otherwise.} \end{cases}$$

The semantics is defined pointwise for each pack p. For statement $x := [a, b]$, we update the octagon $\hat{s}(p)$ of pack p if the pack p contains the variable x; otherwise, the octagon for the current pack is not modified. The actual update is performed by $\hat{R}$. For statement $x := \pm y + [a, b]$, if the pack p does not contain the variable x, we do not update anything. Otherwise, we discern two cases. When p contains the variable y, we can handle the assignment directly by $\hat{R}$. When p does not contain y, we handle the assignment by converting y into its interval value. The function $\mathsf{toInt}_x \in \hat{\mathbb{S}} \to \hat{\mathbb{I}}$ undertakes the conversion.

5.2. Step 2: Finding Definitions and Uses

We now approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. Because the language in this section is pointer-free, simple syntactic method is enough to find them.

The distinguishing feature of sparse relational analysis is that the entities that are defined and used are variable packs, not each variable. From the definition of $\hat{f}_i$, we notice that packs that contain x (denoted by $\mathsf{pack}(x)$) are potentially defined in both assignments:

$$\hat{\mathsf{D}}(i) = \begin{cases} \mathsf{pack}(x) & \mathsf{cmd}(i) = x := [a, b] \\ \mathsf{pack}(x) & \mathsf{cmd}(i) = x := \pm y + [a, b]. \end{cases}$$

We define the use set $\hat{\mathsf{U}}$ as follows:

$$\hat{\mathsf{U}}(i) = \begin{cases} \mathsf{pack}(x) & \mathsf{cmd}(i) = x := [a, b] \\ \mathsf{pack}(x) \cup \mathsf{pack}(y) & \mathsf{cmd}(i) = x := \pm y + [a, b]. \end{cases}$$

It is trivial to check that such $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ satisfy the safety conditions in Definition 3.21.

LEMMA 5.1. *$\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations.*

6. IMPLEMENTATION TECHNIQUES

In this section, we summarize techniques that we used in the implementation of the sparse analyzers (for the C language), which will be evaluated in Section 7. Implementing sparse analysis presents unique challenges regarding construction and management of data dependencies. Because data dependencies for realistic programs can be very complex, it is a key to practical sparse analyzers to efficiently generate data dependencies. We describe the basic algorithm that we used for data dependency generation and discuss two issues that we experienced causing significant performance impacts depending on different implementation choices.

6.1. Generation of Data Dependencies

We use the standard SSA algorithm [Cytron et al. 1991] to generate data dependencies. For C-like language, because control flows of the program are known a priori, our data dependency relation can be simplified to the following:

$$\begin{aligned} i_0 \overset{l}{\rightsquigarrow} i_n \ \ \textit{iff} \ \ & \exists i_0 \ldots i_n \in \mathsf{Paths}, l \in \hat{\mathbb{L}}. \\ & l \in \hat{\mathsf{D}}(i_0) \cap \hat{\mathsf{U}}(i_n) \wedge \forall k \in (0, n). l \notin \hat{\mathsf{D}}(i_k), \end{aligned}$$

where Paths is the set of all paths in the program. Suppose that we have computed $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ for all i. We can generate the data dependencies by propagating each definition $l \in \hat{\mathsf{D}}(i_0)$ along control flows to its use points (where $l \in \hat{\mathsf{U}}(i_n)$) unless l is re-defined, which can be performed with standard def-use chain generation algorithms such as reaching definition analysis or SSA algorithms. We use the SSA generation because it is fast and reduces the size of data dependencies [Wegman and Zadeck 1991].

6.2. Interprocedural Extension

With a semantics-based approach in mind, interprocedural sparse analysis is no more difficult than its intraprocedural counterpart. Designing a method to find safe definitions and uses for semantic functions regarding procedure calls is all that we need for interprocedural extension. For example, consider the language and analysis in Section 4 with procedure calls extended:

$$cmd \to \cdots \mid \mathsf{call}(p_x, e).$$

Command $\mathsf{call}(p_x, e)$ means that procedure p, whose formal parameter is x, is called with actual parameter e. Suppose that the analysis is context insensitive.[3] Then, the abstract semantics for procedure calls is as follows:

$$\hat{f}_i(\hat{s}) = \hat{s}[x \mapsto \hat{\mathcal{E}}(e)(\hat{s})].$$

Simply, the abstract location defined by this semantics is x and uses include the locations that are referenced inside e. Then, data dependencies are generated over the entire program in the same way as in Section 6.1.

However, during the implementation, we noticed that this natural extension does not scale in practice. The main problem was due to unexpected spurious dependencies across procedure boundaries. Consider the following code and suppose that we compute data dependencies for global variable x, which is not used inside procedure h:

```
int f()  {x:=0;^1h();^2a:=x;}
int h()  {...}                          (* no use of x *)
int g()  {x:=1;^3h();^4b:=x;}.
```

Data dependencies for x not only include $1 \overset{x}{\rightsquigarrow} 2$ and $3 \overset{x}{\rightsquigarrow} 4$ but also include spurious dependencies $1 \overset{x}{\rightsquigarrow} 4$ and $3 \overset{x}{\rightsquigarrow} 2$ because of spurious control flow paths via the calls to h:

$$\begin{tikzcd} 1 \arrow[dr, "x"] & & 3 \arrow[dl, "x"'] \\ & h \arrow[dl, "x"] \arrow[dr, "x"'] & \\ 2 & & 4 \end{tikzcd}$$

In real C programs, thousands of global variables exist and procedures are called from many different call sites, which generates an overwhelming number of spurious dependencies. In our experiments, such spurious dependencies made the sparse analysis hardly scalable. Staged pointer analysis algorithm [Hardekopf and Lin 2011] takes this approach, but no performance problem has been reported; we guess that this is because pointer analysis typically ignores nonpointer statements (by sparse evaluation techniques [Choi et al. 1991; Ramalingam 2002]) and the number of pointer variables are a small subset of the entire variables. However, our analyzers trace all semantics of C—that is, value flows of all types including pointers and numbers.

In our approach, we discern the accessed locations by the callee and its transitively called procedures from the rest. For the other locations, we draw dependencies from the call site directly to the return sites. For example, we generate dependencies for the preceding program as follows:

$$\begin{tikzcd} 1 \arrow[dr] \arrow[dd, bend right, "x"'] & & 3 \arrow[dl] \arrow[dd, bend left, "x"] \\ & h \arrow[dl] \arrow[dr] & \\ 2 & & 4 \end{tikzcd}$$

Note that variable x is not propagated to procedure h because h does not use x.

For the interprocedural extension, we use the flow-insensitive analysis (defined in Section 4.2) to pre-resolve function pointers. Because the preanalysis is fairly precise,[4]

[3]Our analyzers that will be evaluated in Section 7 are also context insensitive.

[4]The pointer abstraction of our preanalysis is basically the same with inclusion-based pointer analysis, which is the most precise form of flow-insensitive pointer analysis [Hardekopf and Lin 2007]. In addition,

Table I. Benchmarks

Program	LOC	Functions	Statements	Blocks	maxSCC	AbsLocs
gzip-1.2.4a	7K	132	6,446	4,152	2	1,784
bc-1.06	13K	132	10,368	4,731	1	1,619
tar-1.13	20K	221	12,199	8,586	13	3,245
less-382	23K	382	23,367	9,207	46	3,658
make-3.76.1	27K	190	14,010	9,094	57	4,527
wget-1.9	35K	433	28,958	14,537	13	6,675
screen-4.0.2	45K	588	39,693	29,498	65	12,566
a2ps-4.14	64K	980	86,867	27,565	6	17,684
sendmail-8.13.6	130K	756	76,630	52,505	60	19,135
nethack-3.3.0	211K	2,207	237,427	157,645	997	54,989
vim60	227K	2,770	150,950	107,629	1,668	40,979
emacs-22.1	399K	3,388	204,865	161,118	1,554	66,413
python-2.5.1	435K	2,996	241,511	99,014	723	51,859
linux-3.0	710K	13,856	345,407	300,203	493	139,667
gimp-2.6	959K	11,728	1,482,230	286,588	2	190,806
ghostscript-9.00	1,363K	12,993	2,891,500	342,293	39	201,161

Note: Lines of code (**LOC**) is obtained by running `wc` on the source before preprocessing and macro expansion. **Functions** reports the number of functions in the source code. **Statements** and **Blocks** report the number of statements and basic blocks in our intermediate representation of programs (after preprocessing). **maxSCC** reports the size of the largest strongly connected component in the callgraph. **AbsLocs** reports the number of abstract locations that are generated during the interval domain-based analysis.

the precision loss caused by this approximation of the callgraph would be reasonably small in practice [Milanova et al. 2004].

6.3. Using Binary Decision Diagrams in Representing Data Dependencies

The second practical issue is memory consumption for storing data dependencies. Analyzing real C programs must deal with hundreds of thousands of statements and abstract locations. Thus, naive representations for the data dependencies immediately makes memory problems. For example, in analyzing ghostscript-9.00 (the largest benchmark in Table I), the data dependencies consist of 201K abstract locations spanning over 2.8M statements. Storing such large dependency relation with a naive set-based implementation, which keeps a map ($\in \Delta \times \Delta \rightarrow \mathcal{P}(\hat{\mathbb{L}})$), did not scale. It only worked for programs of moderate sizes with less than 150 KLOC. We solved this problem with binary decision diagrams (BDDs). Fortunately, the data dependency relation is highly redundant, making it a good application of BDDs. For example, $\langle c_1, c_3, l\rangle \in (\rightsquigarrow)$ and $\langle c_2, c_3, l\rangle$ are different but share the common suffix, and $\langle c_1, c_2, l_1\rangle$ and $\langle c_1, c_2, l_2\rangle$ are different but share the common prefix. BDDs can effectively share such common suffixes and prefixes. We treat each relation $\langle c_1, c_2, l\rangle$, by bit encoding each partitioning index (control point) and abstract location, as a Boolean function that is naturally represented by BDDs. This way of using BDDs greatly reduced memory costs. For example, for vim60 (227 KLOC), set-based representation of data dependencies required more than 24GB of memory, but BDD implementation just required 1GB. No particular dynamic variable ordering was necessary in our case.

7. EXPERIMENTS

In this section, we evaluate the sparse nonrelational and relational static analyses designed in Section 4 and Section 5, respectively. The evaluation was performed on

our preanalysis combines numeric analysis and pointer analysis, which further enhances the precision of the pointer analysis [Cousot et al. 2009; Balakrishnan and Reps 2004].

top of SPARROW [Jhee et al. 2008; Oh and Yi 2010, 2011; Oh et al. 2011, 2012], an industrial-strength static analyzer for C programs.

For the nonrelational analysis, we use the interval domain [Cousot and Cousot 1977], a representative nonrelational domain that is widely used in practice [Balakrishnan and Reps 2004; Jung et al. 2005; Blanchet et al. 2003; Cousot et al. 2009; Allamigeon et al. 2006]. For the relational analysis, we use the octagon domain [Miné 2006b], a representative relational domain whose effectiveness is well known in practice [Blanchet et al. 2003; Cousot et al. 2009; Venet and Brat 2004; Lee et al. 2012].

We have analyzed 16 software packages. Table I shows the benchmark programs. The benchmarks are various open source applications, and most of them are from GNU open source projects. The Linux kernel includes only a few drivers (keyboard, power management, block device, and terminal) but includes many other modules such as file system, memory management, ×86 architecture, and so on. The analyses were performed globally (whole-program analysis); the entire program is analyzed starting from procedure main (for Linux, start_kernel). In all analyses, we used handcrafted function stubs for standard library calls. For other unknown procedure calls to external code, we assumed that the procedure returns arbitrary values and has no side effect. Procedures that are unreachable from the main procedure, such as callbacks, are made to be explicitly called from the main procedure. All experiments were done on a Linux 2.6 system running on a single core Intel 3.07GHz box with 24GB of main memory.

7.1. Interval Domain-Based Sparse Analysis

Setting. The baseline analyzer, $\mathsf{Interval}_{\mathsf{base}}$, is the global abstract interpretation engine of SPARROW. The abstract domain of the analysis is an extension of the one defined in Section 4 to support additional C features such as arrays and structures. The analysis abstracts an array by a set of tuples of base address, offset, and size. Abstraction of dynamically allocated arrays is similarly handled, except base addresses are abstracted by their allocation sites. A structure is abstracted by a tuple of base address and set of field locations (the analysis is field sensitive). The fixpoint is computed by a worklist algorithm using the conventional widening operator [Cousot and Cousot 1977] for interval domains. Details of the analysis can be found in Oh et al. [2011]. The analysis is designed to be general purpose: it accepts a full set of (ANSI and GNU) C, including dynamic memory allocation and recursion, which is sometimes not considered in domain-specific analyzers [Blanchet et al. 2003; Miné 2006a; Cousot et al. 2009].

The baseline analyzer is not a straw-man, but much engineering effort has been put to its implementation. It adopts a set of well-known cost reduction techniques in static analysis such as efficient worklist/widening strategies [Bourdoncle 1993] and selective memory operators [Blanchet et al. 2003]. In particular, the analysis exploits the technique of localization [Rinetzky et al. 2005; Yang et al. 2008; Oh et al. 2011], which localizes the analysis so that each code block is analyzed with only the to-be-accessed parts of the input state. We use the access-based technique [Oh et al. 2011; Oh and Yi 2011].

From the baseline, we made $\mathsf{Interval}_{\mathsf{vanilla}}$ and $\mathsf{Interval}_{\mathsf{sparse}}$. $\mathsf{Interval}_{\mathsf{vanilla}}$ is identical to $\mathsf{Interval}_{\mathsf{base}}$ except that $\mathsf{Interval}_{\mathsf{vanilla}}$ does not perform the access-based localization. We compare the performance between $\mathsf{Interval}_{\mathsf{vanilla}}$ and $\mathsf{Interval}_{\mathsf{base}}$ just to check that our baseline analyzer is not a straw-man. $\mathsf{Interval}_{\mathsf{sparse}}$ is the sparse version derived from the baseline. The sparse analysis consists of three steps: preanalysis (to approximate def-use sets), data dependency generation, and actual fixpoint computation. As described in Section 4, we use a flow-insensitive preanalysis. The fixpoint of sparse abstract semantic function is computed by a worklist-based fixpoint algorithm. The analyzers are written in OCaml. We use the BuDDy library [Lind-Nielson n.d.] for the BDD implementation.

Table II. Performance of Interval Analysis

Programs	LOC	Interval$_{\text{vanilla}}$		Interval$_{\text{base}}$			
		Time	Mem	Time	Mem	Spd$\uparrow_1$	Mem$\downarrow_1$
gzip-1.2.4a	7K	772	240	14	65	55×	73%
bc-1.06	13K	1,270	276	96	126	13×	54%
tar-1.13	20K	12,947	881	338	177	38×	80%
less-382	23K	9,561	1,113	1,211	378	8×	66%
make-3.76.1	27K	24,240	1,391	1,893	443	13×	68%
wget-1.9	35K	44,092	2,546	1,214	378	36×	85%
screen-4.0.2	45K	∞	N/A	31,324	3,996	N/A	N/A
a2ps-4.14	64K	∞	N/A	3,200	1,392	N/A	N/A
sendmail-8.13.6	130K	∞	N/A	∞	N/A	N/A	N/A
nethack-3.3.0	211K	∞	N/A	∞	N/A	N/A	N/A
vim60	227K	∞	N/A	∞	N/A	N/A	N/A
emacs-22.1	399K	∞	N/A	∞	N/A	N/A	N/A
python-2.5.1	435K	∞	N/A	∞	N/A	N/A	N/A
linux-3.0	710K	∞	N/A	∞	N/A	N/A	N/A
gimp-2.6	959K	∞	N/A	∞	N/A	N/A	N/A
ghostscript-9.00	1,363K	∞	N/A	∞	N/A	N/A	N/A

Programs	LOC	Interval$_{\text{sparse}}$							
		Dep	Fix	Total	Mem	$\hat{\mathsf{D}}(i)$	$\hat{\mathsf{U}}(i)$	Spd$\uparrow_2$	Mem$\downarrow_2$
gzip-1.2.4a	7K	2	1	3	63	2.4	2.5	5×	3%
bc-1.06	13K	4	3	7	75	4.6	4.9	14×	40%
tar-1.13	20K	6	2	8	93	2.9	2.9	42×	47%
less-382	23K	27	6	33	127	11.9	11.9	37×	66%
make-3.76.1	27K	16	5	21	114	5.8	5.8	90×	74%
wget-1.9	35K	8	3	11	85	2.4	2.4	110×	78%
screen-4.0.2	45K	724	43	767	303	53.0	54.0	41×	92%
a2ps-4.14	64K	31	9	40	353	2.6	2.8	80×	75%
sendmail-8.13.6	130K	517	227	744	678	20.7	20.7	N/A	N/A
nethack-3.3.0	211K	14,126	2,247	16,373	5,298	72.4	72.4	N/A	N/A
vim60	227K	17,518	6,280	23,798	5,190	180.2	180.3	N/A	N/A
emacs-22.1	399K	29,552	8,278	37,830	7,795	285.3	285.5	N/A	N/A
python-2.5.1	435K	9,677	1,362	11,039	5,535	108.1	108.1	N/A	N/A
linux-3.0	710K	26,669	6,949	33,618	20,529	76.2	74.8	N/A	N/A
gimp-2.6	959K	3,751	123	3,874	3,602	4.1	3.9	N/A	N/A
ghostscript-9.00	1,363K	14,116	698	14,814	6,384	9.7	9.7	N/A	N/A

Note: Time (in seconds) and Peak Memory Consumption (in megabytes) of the Various Versions of Analyses. ∞ means the analysis ran out of time (exceeded 24-hour time limit). **Dep** and **Fix** report the time spent during data dependency analysis and actual analysis steps, respectively, of the sparse analysis. **Spd**$\uparrow_1$ is the speed-up of Interval$_{\text{base}}$ over Interval$_{\text{vanilla}}$. **Mem**$\downarrow_1$ shows the memory savings of Interval$_{\text{base}}$ over Interval$_{\text{vanilla}}$. **Spd**$\uparrow_2$ is the speed-up of Interval$_{\text{sparse}}$ over Interval$_{\text{base}}$. **Mem**$\downarrow_2$ shows the memory savings of Interval$_{\text{sparse}}$ over Interval$_{\text{base}}$. $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ show the average size of definition sets and use sets at i, respectively.

Results. Table II gives the analysis time and peak memory consumption of the three analyzers. Because the analyzers share a common front end, we report only the analysis time. For Interval$_{\text{base}}$, the time includes the preanalysis [Oh et al. 2011]. For Interval$_{\text{sparse}}$, **Dep** includes times for preanalysis and data dependency generation. **Fix** represents the time for fixpoint computation of the sparse abstract semantic function.

The results show that Interval$_{\text{base}}$ already has a competitive performance: it is 8 to 55 times faster than Interval$_{\text{vanilla}}$, saving peak memory consumption by 54% to 85%.

$\mathsf{Interval}_{\mathsf{vanilla}}$ scales to 35 KLOC before running out of time limit (24 hours). In contrast, $\mathsf{Interval}_{\mathsf{base}}$ scales to 64 KLOC.

$\mathsf{Interval}_{\mathsf{sparse}}$ is 5 to 110 times faster than $\mathsf{Interval}_{\mathsf{base}}$ and saves memory by 3% to 92%. In particular, the analysis' scalability has been remarkably improved: $\mathsf{Interval}_{\mathsf{sparse}}$ scales to 1.4M LOC, which is an order of magnitude larger than that of $\mathsf{Interval}_{\mathsf{base}}$.

There are some counterintuitive results. First, the analysis time for $\mathsf{Interval}_{\mathsf{sparse}}$ does not strictly depend on program sizes. For example, analyzing emacs-22.1 (399 KLOC) requires 10 hours, taking six times more than analyzing ghostscript-9.00 (1,363 KLOC). This is mainly because some real C programs have unexpectedly large recursive call cycles [Lattner et al. 2007; Yu et al. 2010; Oh and Yi 2011]. Column **maxSCC** in Table I reports the sizes of the largest strongly connected component in the callgraph. Note that some programs (such as nethack-3.3.0, vim60, and emacs-22.1) have a large cycle that contains hundreds or even thousands of procedures. Such big SCCs markedly increase the analysis cost because the large cyclic dependencies among procedures make data dependencies much more complex. Thus, the analysis for gimp-2.6 (959 KLOC) or ghostscript-9.00 (1,363 KLOC), which have little recursion, is even faster than python-2.5.1 (435 KLOC) or nethack-3.3.0 (211 KLOC), which have large recursive cycles.

Second, data dependency generation takes longer time than actual fixpoint computation. For example, data dependency generation for ghostscript-9.00 takes 14,116s, but the fixpoint is computed in 698s. The seemingly unbalanced timing results are partly because of the uses of BDDs in dependency construction. Although BDD dramatically saves memory costs, set operations for BDDs such as addition and removal are noticeably slower than usual set operations. Especially, large programs are more influenced by this characteristic because their data dependency generation is more complex and more BDD operations are involved. However, thanks to the space effectiveness of BDDs, our sparse analysis does not steeply increase memory consumption as program sizes increase.

7.2. Octagon Domain-Based Sparse Analysis

Setting. We implemented octagon domain-based static analyzers $\mathsf{Octagon}_{\mathsf{vanilla}}$, $\mathsf{Octagon}_{\mathsf{base}}$, and $\mathsf{Octagon}_{\mathsf{sparse}}$ by replacing interval domains of SPARROWwith octagon domains. $\mathsf{Octagon}_{\mathsf{base}}$ performs the access-based localization [Oh et al. 2011] in terms of variable packs. $\mathsf{Octagon}_{\mathsf{vanilla}}$ is the same as $\mathsf{Octagon}_{\mathsf{base}}$ except for the localization. $\mathsf{Octagon}_{\mathsf{sparse}}$ is the sparse version of $\mathsf{Octagon}_{\mathsf{base}}$. To represent octagon domains, we used the Apron library [Jeannet and Miné 2009].

In all experiments, we used a syntax-directed packing strategy. Our packing heuristic is similar to Miné's approach [Miné 2006b; Cousot et al. 2009], which groups abstract locations that have syntactic locality. For examples, abstract locations involved in the linear expressions or loops are grouped together. Scope of the locality is limited within each of syntactic C blocks. We also group abstract locations involved in actual and formal parameters, which is necessary to capture relations across procedure boundaries. Large packs whose sizes exceed a threshold (10) were split down into smaller ones.

Results. Whereas $\mathsf{Octagon}_{\mathsf{vanilla}}$ requires an extremely large amount of time and memory, $\mathsf{Octagon}_{\mathsf{base}}$ makes the analysis realistic by leveraging the access-based localization. $\mathsf{Octagon}_{\mathsf{base}}$ is able to analyze 35 KLOC within 5 hours and 10GB of memory. With the localization, analysis speed of $\mathsf{Octagon}_{\mathsf{base}}$ increases by 8 to 9 times and memory consumption decreases by 54% to 62%. Although $\mathsf{Octagon}_{\mathsf{base}}$ saves a lot of memory, the analysis is still not scalable at all. For example, tar-1.13 requires six times more memory than gzip-1.2.4a.

Thanks to our sparse analysis technique, $\mathsf{Octagon}_{\mathsf{sparse}}$ becomes more practical and scales to 130 KLOC within 18 hours and 29GB of memory consumption. $\mathsf{Octagon}_{\mathsf{sparse}}$

Table III. Performance of Octagon Analysis

Programs	LOC	$\mathsf{Octagon}_{\mathsf{vanilla}}$		$\mathsf{Octagon}_{\mathsf{base}}$		Spd$\uparrow_1$	Mem$\downarrow_1$
		Time	Mem	Time	Mem		
gzip-1.2.4a	7K	2,078	2,832	273	1,072	8×	62%
bc-1.06	13K	9,536	6,987	1,065	3,230	9×	54%
tar-1.13	20K	∞	N/A	9,566	5,963	N/A	N/A
less-382	23K	∞	N/A	16,121	8,410	N/A	N/A
make-3.76.1	27K	∞	N/A	17,724	12,771	N/A	N/A
wget-1.9	35K	∞	N/A	15,998	9,363	N/A	N/A
screen-4.0.2	45K	∞	N/A	∞	N/A	N/A	N/A
a2ps-4.14	64K	∞	N/A	∞	N/A	N/A	N/A
sendmail-8.13.6	130K	∞	N/A	∞	N/A	N/A	N/A

Programs	LOC	$\mathsf{Octagon}_{\mathsf{sparse}}$						Spd$\uparrow_2$	Mem$\downarrow_2$
		Dep	Fix	Total	Mem	$\hat{\mathsf{D}}(i)$	$\hat{\mathsf{U}}(i)$		
gzip-1.2.4a	7K	7	14	21	269	13.8	14.5	13×	75%
bc-1.06	13K	20	35	55	358	25.2	31.7	19×	89%
tar-1.13	20K	55	133	188	526	38.3	39.3	51×	91%
less-382	23K	92	340	432	458	42.6	45.4	37×	95%
make-3.76.1	27K	91	240	331	666	51.4	55.7	53×	95%
wget-1.9	35K	107	181	288	646	31.9	32.9	56×	93%
screen-4.0.2	45K	2,452	13,981	16,433	9,199	372.4	376.1	N/A	N/A
a2ps-4.14	64K	296	8,271	8,566	1,996	97.7	99.0	N/A	N/A
sendmail-8.13.6	130K	7,256	57,552	64,808	29,658	467.6	492.3	N/A	N/A

Note: Time (in seconds) and Peak Memory Consumption (in megabytes) of the Various Versions of Analyses.
∞ means the analysis ran out of time (exceeded 24-hour time limit). **Dep** and **Fix** report the time spent during data dependency analysis and actual analysis steps, respectively, of the sparse analysis. **Spd**$\uparrow_1$ is the speed-up of $\mathsf{Interval}_{\mathsf{base}}$ over $\mathsf{Interval}_{\mathsf{vanilla}}$. **Mem**$\downarrow_1$ shows the memory savings of $\mathsf{Interval}_{\mathsf{base}}$ over $\mathsf{Interval}_{\mathsf{vanilla}}$. **Spd**$\uparrow_2$ is the speed-up of $\mathsf{Interval}_{\mathsf{sparse}}$ over $\mathsf{Interval}_{\mathsf{base}}$. **Mem**$\downarrow_2$ shows the memory savings of $\mathsf{Interval}_{\mathsf{sparse}}$ over $\mathsf{Interval}_{\mathsf{base}}$. $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ show the average size of definition sets and use sets, respectively.

is 13 to 56 times faster than $\mathsf{Octagon}_{\mathsf{base}}$ and saves memory consumption by 75% to 95%.

7.3. Discussion

Sparsity. We discuss the relation between performance and sparsity. Column $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ in Tables II and III show how many abstract locations are defined and used for each basic block on average. It clearly shows the key observation in sparse analysis for real programs; only a few abstract locations are defined and used in each program point. For example, the interval domain-based analysis of a2ps-4.14 defines and uses only 0.1% of abstract locations at one program point.

One interesting observation from the experiment results is that the analysis performance depends more on the sparsity than the program size. For instance, even though ghostscript-9.00 is 3.5 times bigger than emacs-22.1 in terms of LOC, ghostscript-9.00 takes 2.6 times less time to analyze. Behind this phenomenon, there is a large difference on sparsity; average $\hat{\mathsf{D}}(i)$ size (and $\hat{\mathsf{U}}(i)$ size) of emacs-22.1 is 30 times bigger than the one of ghostscript-9.00.

Variable packing. For maximal precision, the packing strategy should be more carefully devised for each target program. However, note that our purpose of experiments is to show relative performance of $\mathsf{Octagon}_{\mathsf{sparse}}$ over $\mathsf{Octagon}_{\mathsf{base}}$, and we applied the same packing strategy for all analyzers. Although our packing strategy is not specialized to each program, the packing strategy reasonably groups logically related variables. The average size of packs is 5 to 7 for our benchmarks. Domain-specific packing strategies, such as ones used in Astrée [Miné 2006b] or CGS [Venet and Brat 2004], reports similar results: 3 to 4 [Miné 2006b] or 5 [Venet and Brat 2004].

Precision improvement. When the analysis uses widening, our sparse analysis even improves the analysis precision. Given original nonsparse analysis $\hat{F}$ and its sparse version $\hat{F}_s$, our theoretical framework guarantees that their least fixpoint solutions are equivalent (i.e., $\mathbf{lfp}\hat{F} \equiv \mathbf{lfp}\hat{F}_s$). However, when the abstract domain is of infinite height (e.g., interval domain) and the analysis uses widening [Cousot and Cousot 1977], the sparse analysis is often more precise than its nonsparse counterpart. For instance, in our experiments, we observed that for some programs, the sparse analysis reduces the number of buffer-overrun alarms more than the nonsparse analysis does. For example, the following table reports the number of buffer-overrun alarms from nonsparse and sparse versions of Sparrowfor some benchmark programs:

Program	#buffer-overrun alarms	
	Nonsparse Sparrow	Sparse Sparrow
tar-1.13	968	942
less-382	661	654
make-3.76.1	1,624	1,524

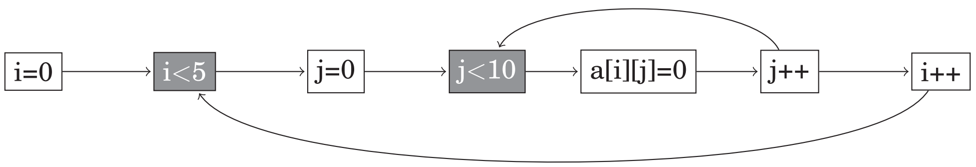
The reason for precision improvement is because widening is less frequently applied in the sparse analysis than in the nonsparse one. In experiments, we found that a typical case happens in nested loops. For instance, consider the following code:

```
int main()
{
  int a[5][10];
  for (i=0;i<5;i++)       // outer loop
    for (j=0;j<10;j++)    // inner loop
      a[i][j] = 0;       // buffer access
}
```

Suppose that we analyze the program with the interval domain and the following simple widening operator for intervals:

$$[l, u] \triangledown [l', u'] = [\text{if } (l' < l) \text{ then } -\infty \text{ else } l, \text{if } (u' > u) \text{ then } +\infty \text{ else } u].$$

Let us first consider the case of the nonsparse analysis, which works over the control flow graph of the program (gray nodes represent headers of flow cycles):



In practice, the widening is usually applied at the headers (gray nodes) of flow cycles. Thus, in a nonsparse interval analysis, the value of variable `i` is widened at both loop headers. In particular, `i` is widened at the header of the inner loop, leading to the analysis result `i` $= [0, +\infty]$. At the first entrance of the inner loop, `i` has interval value $[0, 0]$. At the second iteration, `i` has $[0, 4]$. These analysis results are combined with widening $[0, 0] \triangledown [0, 4] = [0, +\infty]$, which is not strong enough to prove that the buffer access is safe.

On the other hand, the sparse interval analysis does not apply widening to `i` at the inner loop. The sparse analysis works over the data dependencies of the program, which is depicted as follows:

$$\begin{tikzcd}
\boxed{i=0} \arrow[r, "i"] & \boxed{i<5} \arrow[r, "i"] \arrow[drr, "i"] & \boxed{i{+}{+}} \arrow[l, bend right, "i"'] & \boxed{j=0} \arrow[r, "j"] & \boxed{j<10} \arrow[r, "j"] \arrow[dl, "j"'] & \boxed{j{+}{+}} \arrow[l, bend right, "j"'] \\
& & & \boxed{a[i][j]=0} & &
\end{tikzcd}$$

Note that the dependency loops for variable `i` and `j` are completely separated; thus, the value of `i` is not corrupted by the dependency loop for `j`. As a result, the sparse analysis can infer that `i` has $[0, 4]$ at the buffer access statement.

8. RELATED WORK

Existing sparse analysis techniques. Our framework extends the previous sparse analysis framework [Oh et al. 2012] in two ways. Whereas the previous framework only supports “C-like” languages and is applicable to static analysis that uses a particular trace partitioning, our framework is general to support various programming languages (such as higher-order or object-oriented languages) and arbitrary trace partitioning.

Other than Oh et al. [2012], there is no general theory for sparse analysis design. The technique of sparse analysis, which propagates individual abstract values from their definitions to uses, has been developed mostly in the dataflow analysis community [Reif and Lewis 1977; Wegman and Zadeck 1991; Dhamdhere et al. 1992; Hardekopf and Lin 2009, 2011]. Reif and Lewis [1977] developed a sparse analysis algorithm for constant propagation and Wegman et al. [1991] extended it to conditional constant propagation. Dhamdhere et al. [1992] showed how to perform sparse partial redundancy elimination. These algorithms are relatively straightforward because they assume particular dataflow analysis problems for simple pointer-free imperative languages. Sparse analysis with pointers has been recently proposed in efforts to improve the flow-sensitive pointer analysis [Hardekopf and Lin 2009, 2011; Li et al. 2011]. Hardekopf and Lin [2009] presented the semisparse pointer analysis algorithm and showed, for the first time, that flow-sensitive pointer analysis can scale to large code bases (up to 474 KLOC). After that, flow-sensitive pointer analysis becomes scalable even to millions of lines of code via staged sparse analysis techniques [Hardekopf and Lin 2011; Li et al. 2011]. However, these algorithms are also tightly coupled with particular (pointer) analyses, and it is not obvious how to generalize them to arbitrarily semantic analysis. In addition, we showed in Section 3.11 that these two sparse techniques can be explained as instances of our framework.

On the other hand, sparse analysis techniques have not been adequately studied in the semantic-based static analysis community (abstract interpretation). As a result, existing static analyzers (e.g., Astrée [Blanchet et al. 2003], CGS [Venet and Brat 2004], SPARROW [Jhee et al. 2008]) designed by abstract interpretation are all nonsparse. In this article, we present a general sparse analysis design theory on top of abstract interpretation, contributing to the sparse analysis literature in three ways. First, we identify a family of static analysis that can be transformed into its sparse versions while preserving the original precision and soundness. Second, we formally present the framework and prove that the resulting sparse analysis is correct. Previously, the correctness of sparse analysis has been only informally argued (e.g., Hardekopf and Lin [2011]). Third, we show that sparse analysis can be applicable not only to imperative languages but also to arbitrary (e.g., functional) languages. In the past, sparse analysis has been applied only to imperative languages.

Sparse evaluation techniques [Choi et al. 1991; Ramalingam 2002; Cytron and Ferrante 1995; Hind and Pioli 1998] are general, but they take a coarse-grained approach to sparsity. The goal of sparse evaluation is to remove statements whose abstract semantics is identity function. For example, in typical pointer analyses, statements for numerical computation are considered, as identity and sparse evaluation techniques remove those statements before analysis begins. Unlike previous sparse analysis techniques, sparse evaluation techniques such as sparse evaluation graphs [Choi et al. 1991] and compact evaluation graphs [Ramalingam 2002] are general within the dataflow analysis framework [Kam and Ullman 1977]. However, sparse evaluations are coarse grained in that they remove identity semantic functions as a whole but the entire abstract states are still propagated as a unit from program point to program point. Thus, sparse evaluation techniques are not effective when the underlying analysis does not have many identity functions, which is usually the case for static analyses that consider "full" semantics, including numbers and pointers. Our framework provides a method to obtain fine-grained sparse analyses in general settings.

Existing localization techniques [Rinetzky et al. 2005; Might and Shivers 2006; Yang et al. 2008; Oh et al. 2011] are less powerful than our sparse analysis framework. They can be understood as only "spatial" localizations. When analyzing code blocks such as procedure bodies, localization attempts to remove irrelevant parts of abstract memories that will not be used during the analysis. Our sparse analysis subsumes this spatial localization. Our sparse analysis performs additional "temporal" localization too in the sense that adjacent statements S_1 and S_2 need not to be analyzed in order if there are no semantic dependencies between them. Our sparse analysis can be understood as doing both spatial and temporal localizations.

Recently, the idea of sparse analysis has been applied to verification of concurrent programs [Farzan and Kincaid 2012; Farzan et al. 2013]. The authors use a sparse representation (called *dataflow graphs*) of concurrent programs, which simplifies the verification task of unbounded multithreaded programs.

Scalable global static analyzers. Our interval and octagon domain-based analyzers achieve higher scalability (up to 1 MLOC and 130 KLOC, respectively) than the previous general-purpose global analyzers. Zitser et al. [2004] report that PolySpace C Verifier [MathWorks n.d.], a commercial tool for detecting various runtime errors, cannot analyze sendmail because of scalability problems. Both our interval and octagon domain-based analyzers can analyze sendmail. Airac [Jung et al. 2005; Oh 2009], a general-purpose interval domain-based global static analyzer, scales only to 30 KLOC in global analysis. Recently, a significant progress has been reported by [Oh et al. 2011], but it still does not scale over 120 KLOC. Other similar (interval domain-based) analyzers are also not scalable to large code bases [Allamigeon et al. 2006; Balakrishnan and Reps 2004]. Nevertheless, there have been scalable domain-specific static analyzers, like Astrée [Blanchet et al. 2003; Cousot et al. 2009] and CGS [Venet and Brat 2004], which scale to hundreds of thousands of lines of code. However, Astrée targets programs that do not have recursion and backward gotos, which enables a very efficient interpretation-based analysis [Cousot et al. 2009], and CGS is not fully flow sensitive [Venet and Brat 2004]. There are other summary-based approaches [Dillig et al. 2008, 2011] for scalable global analysis, which are independent of our abstract interpretation-based approach.

9. CONCLUSION

9.1. Summary

We have presented a framework for sparse analysis. Given a large class of static analyses defined by abstract interpretation, our framework provides how to transform the analysis into its sparse version while preserving the soundness and precision of the original analysis. We have formally presented the framework, designed two instance analyses for the C language, and experimentally showed that the sparse versions of the instance analyses far exceed the performance of nonsparse versions in a realistic setting.

Our results suggest the following guideline in designing sound and precise, yet scalable, global static analyzers: analysis designers first use the abstract interpretation framework to have a sound and arbitrarily precise global static analyzer. The static analysis in this step is sound and precise but often unscalable. Next, analysis designers use the presented sparse analysis framework to improve the scalability. In contrast to the common sense in static analysis that scalability is obtained by compromising the precision or soundness of the analysis, the resulting sparse analysis still preserves the precision and soundness of the original analysis.

9.2. Open Issues

(1) *Practical implementation for languages other than C*: In this article, we have demonstrated the framework only for the C language. Applying our framework for other languages may be more difficult than that for C. For instance, for dynamic languages such as JavaScript, our simple flow-insensitive preanalysis may not be effective, since too imprecise analyses can take much time for those languages. Designing a preanalysis that is cheap yet precise for other languages remains an open problem.

(2) *Practical implementation with arbitrary trace partitioning*: Although our theoretical framework is generally applicable to arbitrary trace partitioning (e.g., context sensitivity), we demonstrated the effectiveness only for flow-sensitive and context-insensitive analysis. Our framework leaves out details that would need to be solved for a practical implementation of, for example, context-sensitive sparse analysis.

APPENDIX

A. CORRECTNESS PROOF

Let $\hat{F}$ be the semantic function for the baseline analysis (defined in Equation (1)). Let D and U be definitions (Definition 3.10) and uses (Definition 3.12) for $\hat{F}$, respectively. Let $\hat{F}_a$ be the sparse version of $\hat{F}$ (defined in Equation (5)), which is derived using approximated definitions ($\hat{\mathsf{D}}$) and uses ($\hat{\mathsf{U}}$) that satisfy the safety conditions in Definition 3.21. In this appendix, we prove the correctness theorem in Section 3 (Theorem 3.23):

THEOREM A.1 (CORRECTNESS).

$$\forall i \in \Delta . \forall l \in \hat{\mathsf{D}}(i) . (\mathbf{lfp} \hat{F}_a)(i)(l) = (\mathbf{lfp} \hat{F})(i)(l).$$

To prove the theorem, we need auxiliary definitions: helper data dependency, helper abstract semantic function, and the equivalence of fixpoint solutions.

Definition A.2 (*Helper Data Dependency*). Helper data dependency is quadruple relation $(\sim) \subseteq \Delta \times \hat{\mathbb{L}} \times \Delta \times (\Delta \to \hat{\mathbb{S}})$ defined as follows:

$$\begin{array}{rl} i_0 \overset{l}{\sim}_{\hat{\phi}} i_n \;\; \mathit{iff} & \exists i_0 \dots i_n \in \mathsf{Paths}(\hat{\phi}), l \in \hat{\mathbb{L}}. \\ & l \in \hat{\mathsf{D}}(i_0) \wedge \forall k \in (0, n). l \notin \hat{\mathsf{D}}(i_k). \end{array}$$

Note that the notion of helper data dependency $(\sim)$ relaxes the conditions of approximated data dependency $(\rightsquigarrow)$ (Definition 3.20): the helper data dependency does not require that the defined location $l \in \hat{\mathsf{D}}(i_0)$ to be used at i_n. Thus, $(\sim_{\hat{\phi}}) \supseteq (\rightsquigarrow_{\hat{\phi}})$ for all $\hat{\phi}$. Using the helper data dependency, we define the helper abstract semantic function, which is the same as $\hat{F}_a$ but is defined over $(\sim)$ instead of $(\rightsquigarrow)$.

Definition A.3 (*Helper Abstract Semantic Function*).

$$\hat{F}_h(\hat{\phi}) = \lambda i \in \Delta . \hat{f}_i \left(\bigsqcup_{i' \overset{l}{\sim}_{\hat{\phi}} i} \hat{\phi}(i')|_l \right).$$

It is easy to verify that helper abstract semantic function $\hat{F}_h$ is greater than or equal to sparse abstract semantic function $\hat{F}_a$.

LEMMA A.4. $\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_h$.

PROOF. Immediate from $(\rightsquigarrow_{\hat{\phi}}) \subseteq (\sim_{\hat{\phi}})$ □.

Next, we define the notion of equivalence of fixpoint solutions.

Definition A.5 (*Equivalence of Fixpoint Solutions*). We say that $\mathcal{S} \in \Delta \to \hat{\mathbb{S}}$ and $\mathcal{S}' \in \Delta \to \hat{\mathbb{S}}$ are *equivalent*, written $\mathcal{S} \equiv \mathcal{S}'$, if and only if the following two conditions hold:

(1) $\forall i \in \Delta, l \in \hat{\mathsf{D}}(i). \mathcal{S}(i)(l) = \mathcal{S}'(i)(l)$
(2) $(\hookrightarrow_{\mathcal{S}}) = (\hookrightarrow_{\mathcal{S}'})$ □

The first condition says that two solutions are equal modulo $\hat{\mathsf{D}}$. The second condition says that $\mathcal{S}$ and $\mathcal{S}'$ generate the same transition relation. Then, our goal is to prove the following theorem, which is stronger than the correctness theorem.

THEOREM A.6. $\mathbf{lfp}\hat{F} \equiv \mathbf{lfp}\hat{F}_a$.

PROOF. We prove the theorem using sublemmas that we will prove shortly:

$$\begin{array}{rll} \mathbf{lfp}\hat{F} & = \mathbf{lfp}\hat{F}_h & \text{(Lemma A.7 and Lemma A.8)} \\ & \equiv \mathbf{lfp}\hat{F}_a & \text{(Lemma A.9).} \quad \square \end{array}$$

The proof proceeds in two steps, which intuitively show that our notion of definition set ($\hat{\mathsf{D}}$) and use set ($\hat{\mathsf{U}}$) are correct, respectively. The first equality ($\mathbf{lfp}\hat{F} = \mathbf{lfp}\hat{F}_h$) means that the definition set ($\hat{\mathsf{D}}$) is correct: the helper analysis ($\hat{F}_h$), which is only different from $\hat{F}$ in that $\hat{F}_h$ brings each abstract value from its definition points, has indeed the same fixpoint solution as the original analysis ($\hat{F}$). The second equality ($\mathbf{lfp}\hat{F}_h \equiv \mathbf{lfp}\hat{F}_a$) means that the use set ($\hat{\mathsf{U}}$) is correct: the sparse analysis ($\hat{F}_a$), which is only different from $\hat{F}_h$ in that $\hat{F}_a$ considers only the locations in $\hat{\mathsf{U}}$, has the equivalent fixpoint solution as the helper analysis ($\hat{F}_h$).

Now, we prove the lemmas. The first two lemmas prove that the least fixpoint of the original abstract semantic function $\hat{F}$ equals the least fixpoint of the helper abstract semantic function $\hat{F}_h$.

LEMMA A.7. $\mathbf{lfp}\hat{F}_h \sqsubseteq \mathbf{lfp}\hat{F}$.

PROOF. Let $\mathcal{S}$ be $\mathbf{lfp}\hat{F}$. To prove the lemma, it is enough to prove that $\mathcal{S}$ is a post-fixpoint of $\hat{F}_h$ (i.e., $\hat{F}_h(\mathcal{S}) \sqsubseteq \mathcal{S}$):

$$\begin{aligned}
\forall i \in \Delta.\hat{F}_h(\mathcal{S})(i) &= \hat{f}_i\left(\bigsqcup_{i' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i')|_l\right) && (\text{def. of } \hat{F}_h)\\
&\sqsubseteq \hat{f}_i\left(\bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i'')\right) && \left(\hat{f}_i \text{ is mono. \& } \bigsqcup_{i' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i')|_l \sqsubseteq \bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i'')\right)\\
&= \hat{F}(\mathcal{S})(i) && (\text{def. of } \hat{F})\\
&= \mathcal{S}(i). && (\mathcal{S} = \mathbf{lfp}\hat{F})
\end{aligned}$$

It remains to prove the following statement:

$$\forall i \in \Delta. \bigsqcup_{i' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i')|_l \sqsubseteq \bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i''). \tag{6}$$

To prove the statement, it is enough to prove that for all $i' \in \Delta$ such that $i' \overset{l}{\leadsto}_{\mathcal{S}} i$,

$$\mathcal{S}(i')|_l \sqsubseteq \bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i'').$$

By the definition of $(\leadsto)$, $i' \overset{l}{\leadsto}_{\mathcal{S}} i$ implies that there exists a path $i_0 \hookrightarrow_{\mathcal{S}} \cdots \hookrightarrow_{\mathcal{S}} i_n \in$ $\mathsf{Paths}(\mathcal{S})$ such that $i_0 = i'$, $i_n = i$, $l \in \hat{\mathsf{D}}(i_0)$, and $\forall k \in (0, n).l \notin \hat{\mathsf{D}}(i_k)$. For the moment, we claim that $\forall k \in (0, n).\mathcal{S}(i_{k-1})(l) \sqsubseteq \mathcal{S}(i_k)(l)$. Using the claim, the proof proceeds as follows:

$$\begin{aligned}
\mathcal{S}(i')|_l &= \mathcal{S}(i_0)|_l && (i' = i_0)\\
&\sqsubseteq \mathcal{S}(i_1)|_l && (\text{the claim})\\
&\sqsubseteq \cdots\\
&\sqsubseteq \mathcal{S}(i_{n-1})|_l && (\text{the claim})\\
&\sqsubseteq \mathcal{S}(i_{n-1})\\
&\sqsubseteq \bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i_n} \mathcal{S}(i'') && (i_{n-1} \hookrightarrow_{\mathcal{S}} i_n)\\
&= \bigsqcup_{i'' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i''). && (i_n = i)
\end{aligned}$$

Now, we prove the claim:

$$\forall k \in (0, n).\mathcal{S}(i_{k-1})(l) \sqsubseteq \mathcal{S}(i_k)(l).$$

The proof proceeds as follows:

$$\begin{aligned}
\mathcal{S}(i_k)(l) &= \hat{F}(\mathcal{S})(i_k)(l) && (\mathcal{S} = \mathbf{lfp}\hat{F})\\
&= \hat{f}_{i_k}\left(\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i_k} \mathcal{S}(i')\right)(l) && (\text{def. of } \hat{F})\\
&= \left(\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i_k} \mathcal{S}(i')\right)(l) && (l \notin \mathsf{D}(i_k) \text{ from } l \notin \hat{\mathsf{D}}(i_k) \text{ and Lemma 3.11})\\
&\sqsupseteq \mathcal{S}(i_{k-1})(l). && (i_{k-1} \hookrightarrow_{\mathcal{S}} i_k) \quad \square
\end{aligned}$$

LEMMA A.8. $\mathbf{lfp}\hat{F} \sqsubseteq \mathbf{lfp}\hat{F}_h$.

PROOF. Let $\mathcal{S}$ be $\mathbf{lfp}\hat{F}_h$. To prove the lemma, it is enough to prove that $\mathcal{S}$ is a post-fixpoint of $\hat{F}$ (i.e., $\hat{F}(\mathcal{S}) \sqsubseteq \mathcal{S}$):

$$\begin{aligned}
\forall i \in \Delta. \hat{F}(\mathcal{S})(i) &= \hat{f}_i\left(\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i')\right) && (\text{def. of } \hat{F})\\
&\sqsubseteq \hat{f}_i\left(\bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')|_l\right) && \left(\hat{f}_i \text{ is mono. and } \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i') \sqsubseteq \bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')|_l\right)\\
&= \hat{F}_h(\mathcal{S})(i) && (\text{def. of } \hat{F}_h)\\
&= \mathcal{S}(i). && (\mathcal{S} = \mathbf{lfp}\hat{F}_h)
\end{aligned}$$

It remains to prove the following statement:

$$\forall i \in \Delta. \bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i} \mathcal{S}(i') \sqsubseteq \bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')|_l.$$

To prove the statement, it is enough to prove that for all $i' \in \Delta$ such that $i' \hookrightarrow_{\mathcal{S}} i$ and for all $l \in \hat{\mathbb{L}}$,

$$\mathcal{S}(i')(l) \sqsubseteq \bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')(l).$$

We consider two cases, $l \in \hat{\mathsf{D}}(i')$ and $l \notin \hat{\mathsf{D}}(i')$:

—$l \in \hat{\mathsf{D}}(i')$: By the definition of $(\leadsto)$ (Definition A.2), we have $i' \overset{l}{\leadsto}_{\mathcal{S}} i$. Thus,

$$\mathcal{S}(i')(l) \sqsubseteq \bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')(l).$$

—$l \notin \hat{\mathsf{D}}(i')$:

$$\begin{aligned}
\mathcal{S}(i')(l) &= \hat{F}_h(\mathcal{S})(i')(l) && (\mathcal{S} = \mathbf{lfp}\hat{F}_h)\\
&= \hat{f}_{i'}\left(\bigsqcup_{i'' \overset{l'}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')|_{l'}\right)(l) && (\text{def. of } \hat{F}_h)\\
&= \left(\bigsqcup_{i'' \overset{l'}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')|_{l'}\right)(l) && (l \notin \mathsf{D}(i') \text{ from } l \notin \hat{\mathsf{D}}(i') \text{ and } \mathcal{S} \sqsubseteq \mathbf{lfp}\hat{F})\\
&= \left(\bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')(l)\right)\\
&\sqsubseteq \left(\bigsqcup_{i'' \overset{l}{\leadsto}_{\mathcal{S}} i} \mathcal{S}(i'')(l)\right). && (i'' \overset{l}{\leadsto}_{\mathcal{S}} i' \wedge i' \hookrightarrow_{\mathcal{S}} i \wedge l \notin \hat{\mathsf{D}}(i') \implies i'' \overset{l}{\leadsto}_{\mathcal{S}} i)
\end{aligned}$$

Here, we deduce

$$\hat{f}_{i'}\left(\bigsqcup_{i'' \overset{l'}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')|_{l'}\right)(l) = \left(\bigsqcup_{i'' \overset{l'}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')|_{l'}\right)(l)$$

from $l \notin \mathsf{D}(i')$ and $\mathcal{S} \sqsubseteq \mathbf{lfp}\hat{F}$. Note that by Lemma A.7, $\mathcal{S} = \mathbf{lfp}\hat{F}_h \sqsubseteq \mathbf{lfp}\hat{F}$ holds and we have

$$\begin{aligned}
\bigsqcup_{i'' \overset{l'}{\leadsto}_{\mathcal{S}} i'} \mathcal{S}(i'')|_{l'} &\sqsubseteq \bigsqcup_{i'' \overset{l'}{\leadsto}_{(\mathbf{lfp}\hat{F})} i'} (\mathbf{lfp}\hat{F})(i'')|_{l'} && (\mathcal{S} \sqsubseteq \mathbf{lfp}\hat{F})\\
&\sqsubseteq \bigsqcup_{i''' \hookrightarrow_{(\mathbf{lfp}\hat{F})} i'} (\mathbf{lfp}\hat{F})(i'''). && (\text{by } (6))
\end{aligned}$$

Then, Lemma 3.11 applies. In the rest of this section, we will frequently use similar arguments. □

Now, we prove that the fixpoint solutions of the helper abstract semantic function and the sparse abstract semantic function are equivalent.

LEMMA A.9. $\mathbf{lfp}\hat{F}_h \equiv \mathbf{lfp}\hat{F}_a$.

PROOF.

$$\begin{aligned}
\mathbf{lfp}\hat{F}_h &= \hat{F}_h(\mathbf{lfp}\hat{F}_a) && (\text{Lemma A.10})\\
&\equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a) && (\text{Lemma A.14})\\
&= \mathbf{lfp}\hat{F}_a. \quad \square
\end{aligned}$$

Lemma A.10 shows that the fixpoint of helper abstract semantic function $\hat{F}_h$ can be obtained from the fixpoint of sparse abstract semantic function $\hat{F}_a$ by applying $\hat{F}_h$.

LEMMA A.10. $\mathbf{lfp}\hat{F}_h = \hat{F}_h(\mathbf{lfp}\hat{F}_a)$.

PROOF.

—$\mathbf{lfp}\hat{F}_h \sqsupseteq \hat{F}_h(\mathbf{lfp}\hat{F}_a)$: By Lemma A.4, we have

$$\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_h.$$

By applying the monotone function $\hat{F}_h$ on both sides, we have

$$\hat{F}_h(\mathbf{lfp}\hat{F}_a) \sqsubseteq \hat{F}_h(\mathbf{lfp}\hat{F}_h) = \mathbf{lfp}\hat{F}_h.$$

—$\mathbf{lfp}\hat{F}_h \sqsubseteq \hat{F}_h(\mathbf{lfp}\hat{F}_a)$: It is enough to prove that $\hat{F}_h(\mathbf{lfp}\hat{F}_a)$ is a post-fixpoint of $\hat{F}_h$—that is,

$$\hat{F}_h^2(\mathbf{lfp}\hat{F}_a) \sqsubseteq \hat{F}_h(\mathbf{lfp}\hat{F}_a).$$

$$\begin{aligned}
& \mathbf{lfp}\hat{F}_a = \mathbf{lfp}\hat{F}_a \\
\Longrightarrow\ & \hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a) && \text{(Lemma A.11, Lemma A.4, and Lemma A.7)} \\
\Longrightarrow\ & \hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \mathbf{lfp}\hat{F}_a \\
\Longrightarrow\ & \hat{F}_h^2(\mathbf{lfp}\hat{F}_a) = \hat{F}_h(\mathbf{lfp}\hat{F}_a). && \text{(Lemma A.13)} \quad \square
\end{aligned}$$

Lemma A.11 shows that the output of the helper abstract semantic function $\hat{F}_h$ and sparse abstract semantic function $\hat{F}_a$ is equivalent if arguments are equivalent.

LEMMA A.11. *If $\hat{\phi} \equiv \hat{\phi}'$ and $\hat{\phi}, \hat{\phi}' \sqsubseteq \mathbf{lfp}\hat{F}$, then $\hat{F}_h(\hat{\phi}) \equiv \hat{F}_a(\hat{\phi}')$.*

PROOF. We prove the two conditions in Definition A.5.

(1) $\forall i \in \Delta, l \in \hat{\mathsf{D}}(i).\hat{F}_h(\hat{\phi})(i)(l) = \hat{F}_a(\hat{\phi}')(i)(l)$.
Now we prove that $\hat{F}_h(\hat{\phi})(i)(l) = \hat{F}_a(\hat{\phi}')(i)(l)$:

$$\begin{aligned}
\hat{F}_h(\hat{\phi})(i)(l) &= \hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)(l) && \text{(def. of } \hat{F}_h\text{)} \\
&= \hat{f}_i\left(\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)|_{\hat{\mathsf{U}}(i)}\right)(l) && \text{(Def. 3.21, Lemma A.12, } l \in \hat{\mathsf{D}}(i)\text{, and Def. 3.15)} \\
&= \hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'}\right)(l) && \text{(def. of } \rightsquigarrow\text{)} \\
&= \hat{F}_a(\hat{\phi}')(i)(l). && \text{(def. of } \hat{F}_a\text{)}
\end{aligned}$$

The second equality needs more explanation. Note that Definition 3.21 ensures that $\mathbb{U}_{\mathsf{D}(i)}(i) \cup \mathbb{U}_{\hat{\mathsf{D}}(i)\setminus\mathsf{D}(i)}(i) \subseteq \hat{\mathsf{U}}(i)$. By Lemma A.12, $\mathbb{U}_{\mathsf{D}(i)}(i) \cup \mathbb{U}_{\hat{\mathsf{D}}(i)\setminus\mathsf{D}(i)}(i) = \mathbb{U}_{\hat{\mathsf{D}}(i)}(i)$ and hence $\mathbb{U}_{\hat{\mathsf{D}}(i)}(i) \subseteq \hat{\mathsf{U}}(i)$ holds. Then, by Definition 3.15,

$$\hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)\Bigg|_{\hat{\mathsf{D}}(i)} = \hat{f}_i\left(\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)\Bigg|_{\hat{\mathsf{U}}(i)}\right)\Bigg|_{\hat{\mathsf{D}}(i)}$$

holds, from which the desired equality is derived since $l \in \hat{\mathsf{D}}(i)$.

(2) $(\hookrightarrow_{\hat{F}_h(\hat{\phi})}) = (\hookrightarrow_{\hat{F}_a(\hat{\phi}')})$.
It is enough to prove that

$$\forall i \in \Delta. \{i' \in \Delta \mid i \hookrightarrow_{\hat{F}_h(\hat{\phi})} i'\} = \{i' \in \Delta \mid i \hookrightarrow_{\hat{F}_a(\hat{\phi}')} i'\}.$$

By Definitions 3.6 (part (3)) and 3.16, it is enough to prove that

$$\forall i \in \Delta, l \in \hat{\mathsf{U}}(i). \hat{F}_h(\hat{\phi})(i)(l) = \hat{F}_a(\hat{\phi}')(i)(l).$$

We consider two cases:
—$l \in \hat{\mathsf{D}}(i)$: We already showed that $\hat{F}_h(\hat{\phi})(i)(l) = \hat{F}_a(\hat{\phi}')(i)(l)$ holds in this case.
—$l \notin \hat{\mathsf{D}}(i)$:

$$\begin{aligned}
\hat{F}_h(\hat{\phi})(i)(l) &= \hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)(l) && (\text{def. of } \hat{F}_h)\\
&= \left(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'}\right)(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \hat{\phi} \sqsubseteq \mathbf{lfp}\hat{F})\\
&= \left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')(l)\right)\\
&= \left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')(l)\right) && (\hat{\phi} \equiv \hat{\phi}')\\
&= \left(\bigsqcup_{i' \overset{l}{\leadsto}_{\hat{\phi}'} i} \hat{\phi}'(i')(l)\right) && (l \in \hat{\mathsf{U}}(i) \text{ and def. of } \leadsto)\\
&= \left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'}\right)(l)\\
&= \hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'}\right)(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \hat{\phi}' \sqsubseteq \mathbf{lfp}\hat{F})\\
&= \hat{F}_a(\hat{\phi}')(i)(l). && (\text{def. of } \hat{F}_a) \quad \square
\end{aligned}$$

The following lemma states that the use template is distributive over $\cup$:

LEMMA A.12. *For all $A, B \subseteq \mathbb{L}, i \in \Delta$,*

$$\mathbb{U}_{A \cup B}(i) = \mathbb{U}_A(i) \cup \mathbb{U}_B(i).$$

PROOF. ($\subseteq$) Suppose that $l \in \mathbb{U}_{A \cup B}(i)$. By definition, there exists $\hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{(\mathbf{lfp}\hat{F})} i} (\mathbf{lfp}\hat{F})(i')$ such that

$$\hat{f}_i(\hat{s})|_{A \cup B} \neq \hat{f}_i(\hat{s}\backslash_l)|_{A \cup B}.$$

Therefore, there exists $l' \in A \cup B$ such that

$$\hat{f}_i(\hat{s})(l') \neq \hat{f}_i(\hat{s}\backslash_l)(l').$$

If $l' \in A$, then it is an evidence that $l \in \mathbb{U}_A(i)$; otherwise, $l \in \mathbb{U}_B(i)$.

($\supseteq$) Suppose that $l \in \mathbb{U}_A(i)$. By definition, there exists $\hat{s} \sqsubseteq \bigsqcup_{i' \hookrightarrow_{(\mathbf{lfp}\hat{F})} i}(\mathbf{lfp}\hat{F})(i')$ such that

$$\hat{f}_i(\hat{s})|_A \neq \hat{f}_i(\hat{s}\backslash_l)|_A.$$

It is an evidence that $l \in \mathbb{U}_{A\cup B}$. Similarly for B. □

Lemma A.13 shows that the result of applying the helper abstract semantic function $\hat{F}_h$ is invariant up to equivalence.

Lemma A.13. *If $\hat{\phi} \equiv \hat{\phi}'$ then $\hat{F}_h(\hat{\phi}) = \hat{F}_h(\hat{\phi}')$.*

Proof. For all $i \in \Delta$ and $l \in \mathbb{L}$,

$$\begin{aligned}
\hat{F}_h(\hat{\phi})(i) &= \hat{f}_i\left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} \hat{\phi}(i')|_l\right) && \text{(def. of } \hat{F}_h) \\
&= \hat{f}_i\left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')|_l\right) && (\hat{\phi} \equiv \hat{\phi}') \\
&= \hat{F}_h(\hat{\phi}')(i). && \text{(def. of } \hat{F}_h) \quad \square
\end{aligned}$$

Lemma A.14. $\hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a)$.

Proof. By Lemma A.11. Note that $\mathbf{lfp}\hat{F}_a \equiv \mathbf{lfp}\hat{F}_a$ and $\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_h \sqsubseteq \mathbf{lfp}\hat{F}$ by Lemma A.4 and Lemma A.7. □

B. RECONSTRUCTING THE ORIGINAL ANALYSIS RESULTS

Sparse analysis result $\mathbf{lfp}\hat{F}_a$ stores only the abstract values that are defined at each partitioning index, and in the correctness theorem, we compared only the defined abstract values between two fixpoints $\mathbf{lfp}\hat{F}$ and $\mathbf{lfp}\hat{F}_a$. However, using the helper data dependency (Definition A.2), it is easy to check the complete equivalence between the entire original analysis result and the sparse analysis result using the following reconstruction procedure.

Lemma B.1 (Reconstruction). *For all $i \in \Delta$ and $l \in \hat{\mathbb{L}}$,*

$$(\mathbf{lfp}\hat{F})(i)(l) = \begin{cases} (\mathbf{lfp}\hat{F}_a)(i)(l) & l \in \hat{\mathsf{D}}(i) \\ \bigsqcup_{i' \overset{l}{\rightsquigarrow}_{(\mathbf{lfp}\hat{F}_a)} i}(\mathbf{lfp}\hat{F}_a)(i')(l) & l \notin \hat{\mathsf{D}}(i). \end{cases}$$

Proof.

—$l \in \hat{\mathsf{D}}(i)$: By the correctness theorem (Theorem A.1), we have

$$(\mathbf{lfp}\hat{F})(i)(l) = (\mathbf{lfp}\hat{F}_a)(i)(l).$$

—$l \notin \hat{\mathsf{D}}(i)$:

$$\begin{aligned}
(\mathbf{lfp}\hat{F})(i)(l) &= (\mathbf{lfp}\hat{F}_h)(i)(l) && \text{(Lemma A.7 and Lemma A.8)}\\
&= (\hat{F}_h(\mathbf{lfp}\hat{F}_a))(i)(l) && \text{(Lemma A.10)}\\
&= \hat{f}_i\left(\bigsqcup_{i' \overset{l'}{\leadsto}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')|_{l'}\right)(l) && \text{(def. of } \hat{F}_h\text{)}\\
&= \left(\bigsqcup_{i' \overset{l'}{\leadsto}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')|_{l'}\right)(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F})\\
&= \bigsqcup_{i' \overset{l}{\leadsto}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')(l). \quad \square
\end{aligned}$$

In practice, when conducting verification, we do not need to entirely perform the reconstruction procedure. Suppose that we would like to check an assertion at a program point and the assertion involves a variable whose value is missing in the sparse analysis result. In this case, it is enough to perform the reconstruction procedure only for the variable and program point instead of reconstructing the entire original analysis result.

C. A FIXPOINT COMPUTATION STRATEGY

Suppose that we compute a fixpoint of the sparse abstract semantic function

$$\hat{F}_a(\hat{\phi}) = \lambda i \in \Delta.\, \hat{f}_i\left(\bigsqcup_{i' \overset{l}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_l\right).$$

We need an efficient algorithm for computing the fixpoint. A naive fixpoint computation strategy, which rebuilds data dependency $\leadsto_{\hat{\phi}}$ whenever the analysis result $\hat{\phi}$ changes, may be inefficient. In practice, we need a smart algorithm that avoids the recomputation of the data dependency. For instance, we can develop an incremental algorithm that newly builds data dependencies only for part of $\hat{\phi}$ that has been changed. Incremental computation of def-use chains has been studied in the literature (e.g., Choi et al. [1996]).

In this appendix, we present a fixpoint computation strategy that reduces recomputation of data dependencies. Our approach is orthogonal to the incremental approach, and we can combine both approaches in practice. Our algorithm does not update data dependency every time $\hat{\phi}$ changes during the fixpoint computation.

The idea is that we put off updating the data dependency until the intermediate results of the analysis become stable. We slightly modify $\hat{F}_a$ and define the following function:

$$\hat{F}_l(\hat{\phi}, \hat{\phi}') = \lambda i \in \Delta.\, \hat{f}_i\left(\bigsqcup_{i' \overset{l}{\leadsto}_{\hat{\phi}} i} \hat{\phi}'(i')|_l\right).$$

Note that $\hat{F}_l$ uses different arguments in computing analysis results and data dependency: the data dependency $\leadsto$ is computed with $\hat{\phi}$, and the analysis results are computed with $\hat{\phi}'$. Our goal is to compute the fixpoint of $\hat{F}_m$ defined as follows:

$$\begin{aligned}\hat{F}_m(\hat{\phi}) &= \mathbf{lfp}\lambda\hat{\phi}'.\hat{F}_l(\hat{\phi}, \hat{\phi} \sqcup \hat{\phi}') \\ &= \mathbf{lfp}\lambda\hat{\phi}'.\lambda i.\hat{f}_i\left(\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}} i} (\hat{\phi} \sqcup \hat{\phi}')(i')|_l\right).\end{aligned}$$

Computing $\mathbf{lfp}\hat{F}_m = \mathbf{lfp}\lambda\hat{\phi}.\mathbf{lfp}\lambda\hat{\phi}'.\hat{F}_l(\hat{\phi}, \hat{\phi} \sqcup \hat{\phi}')$ consists of nested fixpoint iterations. During the inner fixpoint computation, $\mathbf{lfp}\lambda\hat{\phi}'.\hat{F}_l(\hat{\phi}, \hat{\phi} \sqcup \hat{\phi}')$, the data dependency $\rightsquigarrow_{\hat{\phi}}$ is not recomputed as $\hat{\phi}$ is constant in the inner fixpoint iteration. The data dependency is recomputed only in the outer fixpoint iteration. Thus, if the transition relation ($\hookrightarrow$) is mostly static and only few flows are discovered during the analysis, the data dependency would be recomputed only few times. Lemma C.1 shows that $\mathbf{lfp}\hat{F}_m$ is equal to $\mathbf{lfp}\hat{F}_a$. We first observe two obvious facts on $\hat{F}_l$:

(1) $\hat{F}_m$ and $\hat{F}_l$ are monotone on their arguments.
(2) $\hat{F}_a(\hat{\phi}) = \hat{F}_l(\hat{\phi}, \hat{\phi})$.

LEMMA C.1. $\mathbf{lfp}\hat{F}_m = \mathbf{lfp}\hat{F}_a$.

PROOF. We prove both $\mathbf{lfp}\hat{F}_m \sqsubseteq \mathbf{lfp}\hat{F}_a$ and $\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_m$.

—$\mathbf{lfp}\hat{F}_m \sqsubseteq \mathbf{lfp}\hat{F}_a$:
It is enough to prove that $\hat{F}_m(\mathbf{lfp}\hat{F}_a) \sqsubseteq \mathbf{lfp}\hat{F}_a$.

$$\begin{aligned}\hat{F}_m(\mathbf{lfp}\hat{F}_a) &= \mathbf{lfp}\lambda\hat{\phi}'.\hat{F}_l(\mathbf{lfp}\hat{F}_a, \mathbf{lfp}\hat{F}_a \sqcup \hat{\phi}') && (\text{def. of } \hat{F}_m) \\ &\sqsubseteq \mathbf{lfp}\ \hat{F}_a. && (\hat{F}_l(\mathbf{lfp}\hat{F}_a, \mathbf{lfp}\ \hat{F}_a \sqcup \mathbf{lfp}\ \hat{F}_a) = \mathbf{lfp}\hat{F}_a)\end{aligned}$$

—$\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_m$:
It is enough to prove that $\hat{F}_a(\mathbf{lfp}\hat{F}_m) \sqsubseteq \mathbf{lfp}\hat{F}_m$:

$$\begin{aligned}\mathbf{lfp}\hat{F}_m &= \hat{F}_m(\mathbf{lfp}\hat{F}_m) \\ &= \mathbf{lfp}\lambda\hat{\phi}'.\hat{F}_l(\mathbf{lfp}\hat{F}_m, \mathbf{lfp}\hat{F}_m \sqcup \hat{\phi}').\end{aligned}$$

Hence,

$$\begin{aligned}\mathbf{lfp}\hat{F}_m &= \hat{F}_l(\mathbf{lfp}\hat{F}_m, \mathbf{lfp}\hat{F}_m \sqcup \mathbf{lfp}\hat{F}_m) \\ &= \hat{F}_a(\mathbf{lfp}\hat{F}_m). \quad \square\end{aligned}$$

REFERENCES

Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. 2006. Static analysis of string manipulations in critical embedded C programs. In *Proceedings of the International Symposium on Static Analysis*. 35–51.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in ×86 binary executables. In *Proceedings of the International Conference on Compiler Construction*. 5–23.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*. 196–207.

Francois Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*. 128–141.

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. 1990. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–310.

Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–66.

Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. 1996. Incremental computation of static single assignment form. In *Proceedings of the 6th International Conference on Compiler Construction (CC'96)*. Springer-Verlag, London, UK, 223–237. http://dl.acm.org/citation.cfm?id=647473.727451.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 269–282.

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of Logic and Computation* 2, 4, 511–547.

Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2009. Why does Astre scale up? *Formal Methods in System Design* 35, 3, 229–264. DOI:http://dx.doi.org/10.1007/s10703-009-0089-6

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 84–97.

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

Ron K. Cytron and Jeanne Ferrante. 1995. Efficiently computing ϕ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems* 17, 3, 487–506.

Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. 1992. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, New York, NY, 212–223.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 270–280. DOI:http://dx.doi.org/10.1145/1375581.1375615

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 187–200. DOI:http://dx.doi.org/10.1145/1926385.1926407

Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, New York, NY, 297–308. DOI:http://dx.doi.org/10.1145/2103656.2103693

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2013. Inductive data flow graphs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*. ACM, New York, NY, 129–142. DOI:http://dx.doi.org/10.1145/2429069.2429086

Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 290–299.

Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 226–238.

Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 289–298.

Michael Hind and Anthony Pioli. 1998. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the International Symposium on Static Analysis*. 57–81.

Bertrand Jeannet and Antoine Miné. 2009. APRON: A library of numerical abstract domains for static analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*. 661–667.

Yongin Jhee, Minsik Jin, Yungbum Jung, Deokhwan Kim, Soonho Kong, Heejong Lee, Hakjoo Oh, Daejun Park, and Kwangkeun Yi. 2008. Abstract interpretation + impure catalysts: Our sparrow experience. *Presentation at the Workshop of the 30 Years of Abstract Interpretation*. Available at ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf.

Richard Johnson and Keshav Pingali. 1993. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 78–89.

Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Proceedings of the International Symposium on Static Analysis*. 203–217.

John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7, 3, 305–317.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*.

Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12).* Lecture Notes in Computer Science, Vol. 7148. Springer, 299–314.

Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 343–353.

Jorn Lind-Nielson. n.d. BuDDy, a Binary Decision Diagram Package. Retrieved July 27, 2014, from http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/.

MathWorks. n.d. Polyspace Embedded Software Verification. Retrieved July 27, 2014, from http://www.mathworks.com/products/polyspace/index.html.

Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the European Symposium on Programming.* Lecture Notes in Computer Science, Vol. 3444. Springer, 5–20.

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming*. 13–25.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise and efficient call graph construction for C programs with function pointers. *Journal of Automated Software Engineering* 11, 1, 7–26.

Antoine Miné. 2006a. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*. ACM, New York, NY, 54–63.

Antoine Miné. 2006b. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1, 31–100.

Hakjoo Oh. 2009. Large spurious cycle in global static analyses and its algorithmic mitigation. In *Proceedings of the Asian Symposium on Programming Languages and Systems.* Lecture Notes in Computer Science, Vol. 5904. Springer, 14–29.

Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. 2011. Access analysis-based tight localization of abstract memories. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11).* Lecture Notes in Computer Science, Vol. 6538. Springer, 356–370.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Hakjoo Oh and Kwangkeun Yi. 2010. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software: Practice and Experience* 40, 8, 585–603. DOI:http://dx.doi.org/10.1002/spe.969

Hakjoo Oh and Kwangkeun Yi. 2011. Access-based localization with bypassing. In *Proceedings of the 9th Asian Symposium on Programming Languages and Systems (APLAS'11).* Lecture Notes in Computer Science, Vol. 7078. Springer, 50–65.

Ganesan Ramalingam. 2002. On sparse evaluation representations. *Theoretical Computer Science* 277, 1–2, 119–147.

John H. Reif and Harry R. Lewis. 1977. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 104–118.

Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. 2005. A semantics for procedure local heaps and its abstractions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 296–309.

Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the International Conference on Compiler Construction*. 17–31.

Arnaud Venet and Guillaume Brat. 2004. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 231–242.

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 13, 2, 181–210.

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. 2008. Scalable shape analysis for systems code. In *Proceedings of the International Conference on Computer Aided Verification*. 385–398.

Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*. ACM, New York, NY, 218–229.

Misha Zitser, D. E. Shaw Group, and Tim Leek. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*. ACM, New York NY, 97–106.

Received March 2013; revised October 2013; accepted January 2014