ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2013-014
March 14, 2013

Global Sparse Analysis Framework

Hakjoo Oh, Kihong Heo, Daejun Park, Jeehoon Kang, and Kwangkeun Yi

March 14, 2013

Abstract

In this article we present a general method for achieving global static analyzers that are precise, 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 non-relational 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; we present that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse non-relational and relational static analyses; we 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.

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's 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.

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our approach generalizes the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpretation framework [8, 9, 10] 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 – towards the design of sound, yet scalable global static analyzers. Note that while the abstract interpretation framework [8, 9, 10] 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 preserving its precision. On the other hand, existing sparse analysis techniques [43, 47, 15, 5, 45, 14, 24, 19, 20] 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 [6, 42] are in general settings but instead they take coarse-grained approach to sparsity, where the entire (non-sparse) abstract state is propagated as a unit from program point to program point.

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

In this article, 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 derive its sparse version that preserves the precision of the original analysis. Our framework is general in three ways:
  - It is applicable to static analysis for various programming languages, e.g., imperative languages, functional languages, etc.
  - It is applicable to static analysis with arbitrary semantics properties, e.g., relational analysis, non-relational analysis, etc.
  - It is applicable to static analysis with arbitrary trace partitioning, e.g., context-sensitivity, path-sensitivity, loop unrolling, etc.
- 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 non-sparse version.

Outline Section 2 explains the sparse analysis framework. Section 3 and 4 design sparse non-relational and relational analyses, respectively, based on our framework. Section 5 discusses several issues involved in the implementations. Section 6 presents the experimental studies. Section 7 discusses related work and Section 8 concludes the paper.

2 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 analysis' soundness and precision. In Section 2.2 and 2.3, we define the collecting semantics of the program. In Section 2.4, we specify a family of baseline abstractions that our framework considers. Then, we derive the sparse version of the baseline analysis in the rest sections.

Though 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.

2.1 Notation

We write $\mathcal{P}(S)$ for the power set of S. The domain of f is written $\mathsf{dom}(f)$. Given function $f \in A \to B$, 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 $\mathsf{dom}(f) \setminus \{a\}$, respectively. We write $f[a \mapsto b]$ for function f with the value of a replaced by b. We write $f[a_1 \mapsto b_1, \cdots, a_n \mapsto b_n]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We write $f[\{a_1, \cdots, a_n\} \overset{w}{\mapsto} b]$ for $f[a_1 \mapsto f(a_1) \sqcup b, \cdots, a_n \mapsto f(a_n) \sqcup b]$ (weak update). For all domains, we assume appropriate $\bot$ and $\top$ as well as order $\sqsubseteq$ and join $\sqcup$. In particular, we define $\sqcup, \sqsubseteq, \top, \bot$ for functions in a pointwise fashion, e.g., $f \sqcup g = \lambda x.f(x) \sqcup g(x)$. Given a (potentially infinite) set S, we write S^+ for the set of all finite non-empty 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.

2.2 Programs

We describe a program's semantics as a transition system $(\mathbb{S}, \rightarrow, \mathbb{S}_\iota)$, where $\mathbb{S}$ is the set of states of the program, $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$ 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 (parital) execution sequence, i.e., $\sigma_0 \in \mathbb{S}_\iota \wedge \forall k. \sigma_k \rightarrow \sigma_{k+1}$. We abuse the notion of transition relation $\rightarrow$ for traces, i.e., $\sigma' \rightarrow \sigma \iff \exists s. (\sigma = \sigma' \cdot s) \wedge (\sigma'_\dashv \rightarrow s)$.

2.3 Collecting Semantics

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

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

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

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

2.4 Baseline Abstraction

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

$$\mathcal{P}(\mathbb{S}^+) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \Delta \rightarrow \mathcal{P}(\mathbb{S}^+) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \Delta \rightarrow \hat{\mathbb{S}}$$

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 $(\Delta \rightarrow \mathcal{P}(\mathbb{S}^+))$, where Δ is the set of partitioning indicies).
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 over-approximates the reachable states of the traces.

In Section 2.4.1, we specify the partitioning abstraction: the definitions of (α_1, γ_1) and semantic function $F^\pi \in (\Delta \rightarrow \mathcal{P}(\mathbb{S}^+)) \rightarrow (\Delta \rightarrow \mathcal{P}(\mathbb{S}^+))$. In Section 2.4.2, we will define the final abstract domain $(\Delta \rightarrow \hat{\mathbb{S}})$ and abstract semantic function $\hat{F} \in (\Delta \rightarrow \hat{\mathbb{S}}) \rightarrow (\Delta \rightarrow \hat{\mathbb{S}})$ as a further abstraction of the partitioning abstraction.

2.4.1 Partitioning Abstraction

We first partition the set of traces. Suppose we are given a partitioning function $\pi : \Delta \rightarrow \mathcal{P}(\mathbb{S}^+)$ 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:

$$\mathcal{P}(\mathbb{S}^+) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \Delta \to \mathcal{P}(\mathbb{S}^+).$$

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

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

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 indicies.

Definition 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 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 [32]. One of the conventional partitioning strategies is the control-point partitioning. Suppose 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 indicies 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 above is indeed sound with respect to the collecting semantics.

Lemma 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 [8].

$$\begin{array}{lcll}
\forall \Sigma \in \mathcal{P}(\mathbb{S}^+), & & & \\
(\alpha_1 \circ F)(\Sigma) & = & \alpha_1(\mathbb{S}_\iota \cup \{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_\dashv \to s\}) & \text{(def. of } F) \\
& = & \alpha_1(\mathbb{S}_\iota) \sqcup \alpha_1(\{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_\dashv \to s\}) & (\alpha_1 \text{ is distrib. over } \cup) \\
& = & (\lambda i.(\alpha_1 \mathbb{S}_\iota)(i)) \sqcup (\lambda i.\{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_\dashv \to s\} \cap \pi(i)) & \text{(def. of } \alpha_1) \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup \{\sigma \cdot s \in \pi(i) \mid \sigma \in \Sigma \wedge \sigma_\dashv \to s\} & \text{(def. of } \sqcup) \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i(\Sigma) & \text{(def. of } f_i) \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup (f_i(\Sigma) \cap f_i(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i'))) & \text{(Lemma 4)} \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i(\Sigma \cap (\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} \pi(i'))) & (f_i \text{ is distrib. over } \cap) \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} (\Sigma \cap \pi(i'))) & \text{(set theory)} \\
& = & \lambda i.(\alpha_1 \mathbb{S}_\iota)(i) \cup f_i(\bigcup_{i' \Rightarrow_{(\alpha_1 \Sigma)} i} (\alpha_1 \Sigma)(i')) & \text{(def. of } \alpha_1) \\
& = & F^\pi(\alpha_1 \Sigma) & \text{(def. of } F^\pi) \\
& = & (F^\pi \circ \alpha_1)(\Sigma) &
\end{array}$$

□

Lemma 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:

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

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

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

□

2.4.2 State Abstraction

Next, we abstract the partitioned collecting semantics by abstracting each partition's traces into an abstract state. Suppose 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, i.e.,

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

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 non-relational abstract domains, such as intervals [8], are members of this family. Furthermore, the family also covers some numerical, relational domains. Practical relational analyses exploit *packed* relationality [11, 36, 46, 3]; the abstract domain is of form $\mathit{Packs} \to \hat{\mathbb{R}}$ 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 ($\hat{\mathbb{V}}$). Examples of the numerical constraints are domains of octagons [36] and polyhedrons [12].

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

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

where $\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ (Definition 5) is an abstract semantic function for partitioning index i and $(\hookrightarrow_{\hat{\phi}}) \subseteq \Delta \times \Delta$ (Definition 6) is an abstract transition relation.

Definition 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. $\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}}. \hat{s} \sqsubseteq \hat{s}' \implies \hat{f}_i(\hat{s}) \sqsubseteq \hat{f}_i(\hat{s}')$
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 indicies.

Definition 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. $(\hookrightarrow_{\alpha_2(\phi)}) \supseteq (\Rightarrow_\phi) = \{(i', i) \in \Delta \times \Delta \mid \sigma' \in \phi(i') \wedge \sigma' \to \sigma \wedge \sigma \in \pi(i)\}$
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 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 [8].

$$\begin{array}{lcll}
\multicolumn{4}{l}{\forall \phi \in \Delta \to \mathcal{P}(\mathbb{S}^+), i \in \Delta,} \\
(\alpha_2 \circ F^\pi)(\phi)(i) & = & \alpha_{\mathbb{S}}((\alpha_1 \mathbb{S}_\iota)(i) \sqcup f_i(\bigcup_{i' \Rightarrow_\phi i} \phi(i'))) & \text{(def. of } \alpha_2 \text{ and } F^\pi) \\
& = & \alpha_{\mathbb{S}}((\alpha_1 \mathbb{S}_\iota)(i)) \sqcup (\alpha_{\mathbb{S}} \circ f_i)(\bigcup_{i' \Rightarrow_\phi i} \phi(i')) & (\alpha_{\mathbb{S}} \text{ is distributive}) \\
& \sqsubseteq & (\hat{f}_i \circ \alpha_{\mathbb{S}})(\bigcup_{i' \Rightarrow_\phi i} \phi(i')) & \text{((2) and (3) of Def. 5)} \\
& = & \hat{f}_i(\alpha_{\mathbb{S}}(\bigcup_{i' \Rightarrow_\phi i} \phi(i')) & \\
& = & \hat{f}_i(\bigsqcup_{i' \Rightarrow_\phi i} \alpha_2(\phi)(i')) & (\alpha_{\mathbb{S}} \text{ is distrib. \& def. of } \alpha_2) \\
& \sqsubseteq & \hat{f}_i(\bigsqcup_{i' \hookrightarrow_{\alpha_2(\phi)} i} \alpha_2(\phi)(i')) & \text{((1) of Def. 6)} \\
& = & (\hat{F} \circ \alpha_2)(\phi)(i) & \text{(def. of } \hat{F})
\end{array}$$

□ □

Lemma 8 (Soundness). *$\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$ where $\alpha = \alpha_2 \circ \alpha_1$.*

Proof. By Lemma 3, Lemma 7, and monotonicity of α_2. □

2.5 Toward Sparse Analysis

The abstract semantic function given in (1) may propagate some abstract values unnecessarily between partitioning indicies. For example, suppose that we analyze 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 abstract states to the statement, only those for variables x and y are enough. However, the function given in (1) blindly propagates the whole 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, though our result is generally applicable to arbitrary static analyses and languages.

Example 9. *We design a simple pointer analysis for a small subset of C. Suppose 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:*

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

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 that x points to, 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 we use the control-point partitioning (Section 2.4.1). Then, the abstract semantic function is defined as follows:

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

where the abstract domain is a map from control point to abstract states, i.e., $\hat{\phi} \in \mathbb{C} \to \hat{\mathbb{S}}$. 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

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

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 $\hat{f}_i \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ 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 $\hat{\mathcal{E}} \in e \to \hat{\mathbb{S}} \to \mathcal{P}(\mathit{Var})$ 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; and (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 the abstract values of variables in $\hat{s}(x)$. □

2.6 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 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, i.e., (let $\mathcal{S} = \mathbf{lfp}\hat{F}$)*

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

□

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

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

Proof. Immediate from the dual statement of Definition 10. □

Note that the notion of definition set is 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 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, i.e., (let $\mathcal{S} = \mathbf{lfp}\hat{F}$)

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

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

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

□

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 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 13. *Suppose that we analyze the following program with the analysis designed in Example 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 9, abstract semantic function $\hat{f}_i$ for each control point i is as follows:

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

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

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

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 generate the value of y ($\mathsf{D}(⑫)$). □

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

Definition 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, i.e., (let $\mathcal{S} = \mathbf{lfp}\hat{F}$)*

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

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 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 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 paper but necessary in the correctness proof (Appendix A). Similarly, we assume that $\hookrightarrow$ satisfies the following property.

Definition 16 (Well-formed Abstract Transition Relation)**.** *We say that abstract transition relation $\hookrightarrow$ is well-formed if*

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

□

2.7 Data Dependencies

Once we identified definition and use sets at each partitioning index, we can discover data dependencies of abstract semantic function $\hat{F}$ between two partitioning indicies. 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. Formal definition of data dependency is given below.

Definition 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:*

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

where $\mathsf{Paths}(\hat{\phi})$ is the set of all paths created by transition relation $\hookrightarrow_{\hat{\phi}}$: a path $p = p_0p_1\cdots p_n$ is a sequence of partitioning indicies such that $p_0 \hookrightarrow_{\hat{\phi}} p_1 \hookrightarrow_{\hat{\phi}} \cdots \hookrightarrow_{\hat{\phi}} p_n$, then,

$$\mathsf{Paths}(\hat{\phi}) = \mathbf{lfp}\lambda P.\{i_0 i_1 \mid i_0 \hookrightarrow_{\hat{\phi}} i_1\} \cup \{p_0p_1\cdots p_n i \mid p \in P \ \land \ p_n \hookrightarrow_{\hat{\phi}} i\}.$$

□

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 indicies i_0 and i_n on location l.

Example 18. *In the program presented in Example 13, we can find two data dependencies, ⑩ $\overset{x}{\rightsquigarrow}$ ⑪ and ⑪ $\overset{x}{\rightsquigarrow}$ ⑫ as graphically depicted as follows:*

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

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

2.8 Sparse Abstract Semantics Function

Using data dependency, we can define 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 is given below, is the same as the original one (1) except that it propagates abstract values along the data dependency, not along the transition relation:

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

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 sparse abstract semantic function is the same as the one of original analysis.

Theorem 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 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 the sparse analysis result does not contain the entries unnecessary for its computation. In case we want to check, after the analysis, some properties involving abstract values that are not defined, it is always possible to construct the entire $\mathbf{lfp}\hat{F}$ from $\mathbf{lfp}\hat{F}_s$ (see Appendix B).

2.9 Sparse Analysis with Approximated Data Dependency

Sparse analysis designed until Section 2.8 is not practical. The definitions of D and U are purely mathematical but non-constructive, 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 2.9.1. Suppose $\hat{\mathsf{D}}$ 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 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:*

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

□

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:

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

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

2.9.1 Conditions for Safe Approximations

In order 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 over-approximations 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 over-approximation. Regarding the second condition, the following example illustrates what happens when there exists an abstract location which is used to generate spurious definitions but is not included in the approximated use set.

Formally, safe approximations of definition and use sets are:

Definition 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. $\hat{\mathsf{U}}(i) \supseteq \mathbb{U}_{(\hat{\mathsf{D}}(i) \setminus \mathsf{D}(i))}(i)$

□

Example 22. *Suppose that we analyze the following program with the original analysis designed in Example 9:*

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

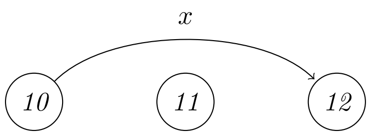
Suppose further that the points-to set for pointer p at ⑪ is $\{y\}$ during the original analysis. Then, according to the analysis definition in Example 9, abstract semantic function $\hat{f}_i$ for each control point i is as follows:

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

Then, definition sets and use sets are as follows:

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

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. Note that, however, not all over-approximations make the sparse analysis safe.

The following is one example of an over-approximations of the above D *and* U*, yet unsafe ones.*

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

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

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

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 21): we adjust $\hat{\mathsf{U}}$ *to include locations that are necessary to generate values of spurious definitions. For our example, we make* $\hat{\mathsf{U}}(\text{⑪})$ *include the spurious definition of* x*:*

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

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

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

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 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 this pre-constructed 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 lemma states:

Theorem 23 (Correctness)**.** *Suppose 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. □

2.10 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 uses of the definition. We can express this def-use chain relation $\frown$ as follows:

Definition 24 (Def-Use Chains)**.**

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

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 $\hat{\mathsf{D}}$. While 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.

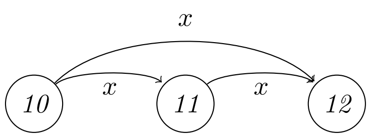
Example 25. *Suppose that we analyze the following program with the original analysis designed in Example 9:*

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

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 that, for sparse analysis, we use the following approximated definition and use sets during the def-use path construction:

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

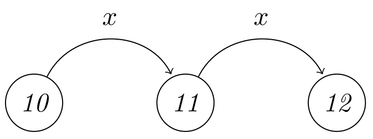
With these information, the conventional def-use chains (Definition 24) become as follows:



$$\begin{tikzcd} 10 \arrow[r, bend left, "x"] \arrow[rr, bend left=40, "x"] & 11 \arrow[r, bend left, "x"] & 12 \end{tikzcd}$$

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 20) builds the def-use paths as follows:



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

Note that there is no join points in the above data dependencies: the main analysis along the above data dependencies does not degrade the analysis precision at ⑫. □

2.11 Existing Sparse Analyses as Instances

In this subsection, we show that recent two sparse analysis techniques [19, 20] can be understood as specific instances of our framework that approximate the definition and use sets in certain ways.

Non-Sparse Analysis Before discussing those two existing sparse techniques, we first show a simple example that non-sparse 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:

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

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 non-sparse analysis, which brings the full abstract state (the abstract values of all locations) along all partitioning indicies.

Semi-Sparse Analysis The challenge of sparse analysis in general is that the definition and use sets are not available prior to the analysis. The semi-sparse technique [19] 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 non-sparse analysis on the other variables. This technique is conceptually identical to our sparse analysis with the following $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$:

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

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 [20] solves the challenge of sparse analysis by employing a pre-analysis and computing conservative def-use information. This idea is formalized in our framework as follows. First, we compute an over-approximation $\hat{X}$ of $\mathbf{lfp}\hat{F}$, i.e., $\hat{X} \sqsupseteq \mathbf{lfp}\hat{F}$, by a pre-analysis. Next, we approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ using $\hat{X}$:

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

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

2.12 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 non-sparse static analysis based on abstract interpretation framework. Note that the abstract domain should be a member of the family explained in Section 2.4.

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 21).

In the following sections, we present two sparse analysis design examples: sparse non-relational analysis (Section 3) and sparse relational analysis (Section 4) for C-like languages. In Section 6, we evaluate the performance of these analyses.

3 Designing Sparse Non-Relational Analysis

In this section, we present an example on designing sparse analysis for non-relational numeric analyses for C-like imperative languages. Following Section 2.12, we first define a non-sparse analysis and then show how to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ that satisfy the safe approximation conditions (Definition 21). The sparse analysis designed in this section is the core of our interval domain-based static analyzer, called $\mathsf{Interval}_{\mathsf{sparse}}$, which will be evaluated in Section 6.

3.1 Step 1: Designing Non-sparse 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 that x points to. An assume command $\{\!\!\{ x < n \}\!\!\}$ makes the program continue only when the condition evaluates to true.

Abstract Domain We consider an analysis that over-approximates 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 non-relational abstract state such that $\mathcal{P}(\mathbb{S}^+) \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$:

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

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 non-relational 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:

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

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.[1] $\{\!\!\{x < n\}\!\!\}$ confines the interval value of x according to the condition. $\hat{\mathcal{E}} \in e \to \hat{\mathbb{S}} \to \hat{\mathbb{V}}$ 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 above 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.

3.2 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 part 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 pre-analysis. Let $\hat{\mathbb{D}}_{pre}$ and $\hat{F}_{pre}$ be the domain and semantic function of such a pre-analysis, which satisfies the following two conditions:

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

By abstract interpretation framework [8, 9, 10], such a pre-analysis is guaranteed to be conservative, i.e., $\alpha_{pre}(\mathbf{lfp}\hat{F}) \sqsubseteq \mathbf{lfp}\hat{F}_{pre}$. As an example, in experiments (Section 6), 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 pre-analysis result. Let $\hat{s}_{pre} \in \hat{\mathbb{S}}$ be the pre-analysis 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 pre-analysis: because we cannot have the input state $\hat{s}$ prior to the analysis, we instead use

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

its conservative abstraction $\hat{s}_{pre}$. Such $\hat{\mathsf{D}}$ satisfies the safe approximation condition (Definition 21), because we collect all potentially defined locations, pre-analysis 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 $\hat{\mathcal{E}}$.

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

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 it includes the entire $\hat{\mathsf{D}}(i)$, it is easy to verify that $\hat{\mathsf{U}}(i)$ satisfies the conditions in Definition 21.

Lemma 26. *$\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations.*

4 Designing Sparse Relational Analysis

In this section, we design a sparse relational analysis. We define a family of relational analyses (Section 4.1) and show a safe approximation of definitions $\hat{\mathsf{D}}$ and uses $\hat{\mathsf{U}}$ for the analysis (Section 4.2). In this

We consider *packed* relational analysis [3, 36]. A pack is a semantically related set of variables. We assume a set of variable packs, $\mathit{Packs} \subseteq \mathcal{P}(\mathit{Var})$ such that $\bigcup \mathit{Packs} = \mathit{Var}$, are given by users or a pre-analysis [36, 11]. 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 the variable packs that contain x may be defined and used at the same time, while only variable x may be defined and not used in non-relational analysis. As a result, data dependencies are also defined in terms of variable packs. We denote a pack of variables $x_1, \cdots, x_n$ as $\langle\!\langle x_1, \cdots, x_n \rangle\!\rangle$.

4.1 Step 1: Designing Non-sparse Analysis

We consider a packed relational analysis based on the octagon abstract domain [36]. This is for the clarify of the presentation and the overall idea is applicable to other relational domains such as the polyhedron [12].

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 non-deterministic 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.

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

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 [36]. We do not consider pointers: including pointers in the language does not require novelty but verbosity. We focus only on the differences between non-relational and relational sparse analysis designs.

Abstract Domain We consider an analysis that computes an over-approximation 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:

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

We suppose the relational domain ($\hat{\mathbb{R}}$) is equipped with the following operators:

- $\mathsf{toInt}_x \in \hat{\mathbb{R}} \rightarrow \hat{\mathbb{I}}$: 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)
- $\hat{R} \in cmd \rightarrow \hat{\mathbb{R}} \rightarrow \hat{\mathbb{R}}$: a semantic function of the relational domain for commands.

For the octagon domain, the definitions of these operators are available in [36]. Regarding the projection function, we suppose that a projection function $\delta_x \in \hat{\mathbb{S}} \rightarrow \hat{\mathbb{I}}$ for the packed relational domain ($\hat{\mathbb{S}}$) is given, which satisfies the following condition:

$$\forall \hat{s} \in \hat{\mathbb{S}}.\delta_x(\hat{s}) \sqsupseteq \alpha_{\hat{\mathbb{I}}}(\textstyle\bigcap_{p \in \mathsf{pack}(x)} \{s(x) \mid s \in \gamma_{\hat{\mathbb{R}}}(\hat{s}(p))\})$$

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

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

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

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

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

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 $\delta_x \in \hat{\mathbb{S}} \to \hat{\mathbb{I}}$ undertakes the conversion.

4.2 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 21.

Lemma 27. *$\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations.*

5 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 6. 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 we used for data dependency generation, and discuss two issues that we experienced significant performance impacts depending on different implementation choices.

5.1 Generation of Data Dependencies

We use the standard SSA algorithm [13] to generate data dependencies. For C-like language, because control flows of the program is known a priori, our data dependency relation can be simplified to the following:

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

where Paths is the set of all paths in the program. Suppose 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 [47].

5.2 Interprocedural Extension

With 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 3 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 the analysis is context-insensitive.[2] 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 Section 5.1.

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 we compute data dependencies for global variable `x`, which is not used inside procedure `h`:

```
int f() { x:=0;^1 h(); ^2a:=x;}
int h() { ...  }                          (* no use of x *)
int g() { x:=1;^3 h(); ^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 [20] takes this approach but no performance problem was reported; we guess this is because pointer analysis typically ignores non-pointer statements (by sparse evaluation techniques [6, 42]) and number of pointer variables are a small subset of the entire variables. However, our analyzers trace all semantics of C, i.e., 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 rest locations, we draw dependencies from the call-site directly to the return sites. For example, we generate dependencies for the above program as follows:

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

[2]Our analyzers that will be evaluated in Section 6 is also context-insensitive.

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 3.2) to pre-resolve function pointers. Because the pre-analysis is fairly precise[3], the precision loss caused by this approximation of the callgraph would be reasonably small in practice [34].

5.3 Using BDDs 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 1), the data dependencies consist of 201 K abstract locations spanning over 2.8 M statements. Storing such large dependency relation with a naive set-based implementation, which keeps a map ($\in \Delta \times \Delta \to \mathcal{P}(\hat{\mathbb{L}})$), did not scale. It only worked for programs of moderate sizes 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 24 GB of memory but BDD-implementation just required 1 GB. No particular dynamic variable ordering was necessary in our case.

6 Experiments

In this section, we evaluate the sparse non-relational and relational static analyses designed in Section 3 and Section 4, respectively. The evaluation was performed on top of Sparrow [23, 25, 40, 38, 41], an industrial-strength static analyzer for C programs.

For the non-relational analysis, we use the interval domain [8], a representative non-relational domain that is widely used in practice [2, 25, 3, 11, 1]. For the relational analysis, we use the octagon domain [36], a representative relational domain whose effectiveness is well-known in practice [3, 11, 46, 28].

We have analyzed 16 software packages. Table 1 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, x86 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 of Intel 3.07 GHz box with 24 GB of main memory.

6.1 Interval Domain-based Sparse Analysis

[3]The pointer abstraction of our pre-analysis is basically the same with inclusion-based pointer analysis, which is the most precise form of flow-insensitive pointer analysis [18]. In addition, our pre-analysis combines numeric analysis and pointer analysis, which further enhances the precision of the pointer analysis [11, 2].

Table 1: Benchmarks: lines of code (**LOC**) is obtained by running `wc` on the source before preprocessing and macro expansion. **Functions** reports the number of functions in 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 .

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

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 3 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 array is similarly handled except that 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 [8] for interval domain. Details of the analysis can be found in [38]. The analysis is designed to be general purpose: it accepts full set of (ANSI and GNU) C, including dynamic memory allocation and recursion, which is sometimes not considered in domain-specific analyzers [3, 35, 11].

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 [4] and selective memory operators [3]. In particular, the analysis exploits the technique of localization [44, 48, 38], 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 [38, 41].

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: pre-analysis (to approximate def-use sets), data dependency generation, and actual fixpoint computation. As described in Section 3, we use a flow-insensitive pre-analysis. 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 [30] for BDD implementation.

Table 2: Performance of interval analysis: 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** reports 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 at i, respectively.

Programs	**LOC**	$\mathsf{Interval}_{\mathsf{vanilla}}$		$\mathsf{Interval}_{\mathsf{base}}$		**Spd**$\uparrow_1$	**Mem**$\downarrow_1$
		Time	**Mem**	**Time**	**Mem**		
gzip-1.2.4a	7K	772	240	14	65	55 x	73 %
bc-1.06	13K	1,270	276	96	126	13 x	54 %
tar-1.13	20K	12,947	881	338	177	38 x	80 %
less-382	23K	9,561	1,113	1,211	378	8 x	66 %
make-3.76.1	27K	24,240	1,391	1,893	443	13 x	68 %
wget-1.9	35K	44,092	2,546	1,214	378	36 x	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**	$\mathsf{Interval}_{\mathsf{sparse}}$						**Spd**$\uparrow_2$	**Mem**$\downarrow_2$
		Dep	**Fix**	**Total**	**Mem**	$\hat{\mathsf{D}}(i)$	$\hat{\mathsf{U}}(i)$		
gzip-1.2.4a	7K	2	1	3	63	2.4	2.5	5 x	3 %
bc-1.06	13K	4	3	7	75	4.6	4.9	14 x	40 %
tar-1.13	20K	6	2	8	93	2.9	2.9	42 x	47 %
less-382	23K	27	6	33	127	11.9	11.9	37 x	66 %
make-3.76.1	27K	16	5	21	114	5.8	5.8	90 x	74 %
wget-1.9	35K	8	3	11	85	2.4	2.4	110 x	78 %
screen-4.0.2	45K	724	43	767	303	53.0	54.0	41 x	92 %
a2ps-4.14	64K	31	9	40	353	2.6	2.8	80 x	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

Table 3: Performance of octagon analysis: 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** reports 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.

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 x	62 %
bc-1.06	13K	9,536	6,987	1,065	3,230	9 x	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 x	75 %
bc-1.06	13K	20	35	55	358	25.2	31.7	19 x	89 %
tar-1.13	20K	55	133	188	526	38.3	39.3	51 x	91 %
less-382	23K	92	340	432	458	42.6	45.4	37 x	95 %
make-3.76.1	27K	91	240	331	666	51.4	55.7	53 x	95 %
wget-1.9	35K	107	181	288	646	31.9	32.9	56 x	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

Results Table 2 gives the analysis time and peak memory consumption of the three analyzers. Because the analyzers share a common frontend, we report only the analysis time. For $\mathsf{Interval}_{\mathsf{base}}$, the time includes the pre-analysis [38]. For $\mathsf{Interval}_{\mathsf{sparse}}$, **Dep** includes times for pre-analysis and data dependency generation. **Fix** represent the time for fixpoint computation of the sparse abstract semantic function.

The results show that $\mathsf{Interval}_{\mathsf{base}}$ already has a competitive performance: it is faster than $\mathsf{Interval}_{\mathsf{vanilla}}$ by 8–55x, saving peak memory consumption by 54–85%. $\mathsf{Interval}_{\mathsf{vanilla}}$ scales to 35 KLOC before running out of time limit (24 hours). In contrast, $\mathsf{Interval}_{\mathsf{base}}$ scales to 111 KLOC.

$\mathsf{Interval}_{\mathsf{sparse}}$ is faster than $\mathsf{Interval}_{\mathsf{base}}$ by 5–110x and saves memory by 3–92%. 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 [27, 49, 41]. Column **maxSCC** in Table 1 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 few 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 14,116 s but the fixpoint is computed in 698 s. This seemingly unbalanced timing results are partly because of the uses of BDDs in dependency construction. While 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.

6.2 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 SPARROW with octagon domains. $\mathsf{Octagon}_{\mathsf{base}}$ performs the access-based localization [38] 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 domain, we used the Apron library [22].

In all experiments, we used a syntax-directed packing strategy. Our packing heuristic is similar to Miné's approach [36, 11], 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 While $\mathsf{Octagon}_{\mathsf{vanilla}}$ requires extremely large amount of time and memory space but $\mathsf{Octagon}_{\mathsf{base}}$ 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 8x–9x and memory consumption decreases by 54%–

62%. Though $\mathsf{Octagon}_{\mathsf{base}}$ saves a lot of memory, the analysis is still not scalable at all. For example, tar-1.13 requires 6 times more memory than gzip-1.2.4a.

Thanks to sparse analysis technique, $\mathsf{Octagon}_{\mathsf{sparse}}$ becomes more practical and scales to 130 KLOC within 18 hours and 29 GB of memory consumption. $\mathsf{Octagon}_{\mathsf{sparse}}$ is 13–56x faster than $\mathsf{Octagon}_{\mathsf{base}}$ and saves memory consumption by 75%–95%.

6.3 Discussion

Sparsity We discuss the relation between performance and sparsity. Column $\hat{\mathsf{D}}(i)$ and $\hat{\mathsf{U}}(i)$ in Table 2 and Table 3 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 is more dependent 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, 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. Though our packing strategy is not specialized to each program, the packing strategy reasonably groups logically related variables. The average size of packs is 5–7 for our benchmarks. Domain-specific packing strategies, such as ones used in Astrée [36] or CGS [46], reports the similar results: 3–4 [36] or 5 [46].

7 Related Work

7.1 Previous Sparse Analysis Techniques

Our framework extends the previous sparse analysis framework [39] in two ways. While 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 [39], 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 [43, 47, 15, 19, 20]. Reif and Lewis developed a sparse analysis algorithm for constant propagation [43] and Wegman et al. extended it to conditional constant propagation [47]. Dhamdhere et al. showed how to perform sparse partial redundancy elimination [15]. 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 [19, 20, 29]. Hardekopf et al. presented the semi-sparse pointer analysis algorithm [19] 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 [20, 29]. 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 2.11 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 [3], CGS [46], Sparrow [23]) designed by abstract interpretation are all non-sparse. In this article, we present a general sparse analysis design theory on top of the abstract interpretation, contributing to the sparse analysis literature in three ways: (1) We identify a family of static analysis that can be transformed into its sparse versions while preserving the original precision and soundness; (2) 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., [20]); (3) We show that sparse analysis can be applicable not only to imperative languages but also to arbitrary (e.g., functional) languages. Previously, sparse analysis has been applied only to imperative languages.

Sparse evaluation techniques [6, 42, 14, 21] are general but they take coarse-grained approach to sparsity. The goal of sparse evaluation is to remove statements whose abstract semantics has identity transference. 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 [6] and compact evaluation graphs [42] are general within the dataflow analysis framework [26]. 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 [44, 33, 48, 38] 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 part 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 is no semantic dependencies between them. Our sparse analysis can be understood as doing both spatial and temporal localizations.

7.2 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. [50] report that PolySpace C Verifier [31], a commercial tool for detecting various runtime errors, cannot analyze sendmail because of scalability problem. Both our interval and octagon domain-based analyzers can analyze sendmail. Airac [25, 37], 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. [38], but it still does not scale over 120 KLOC. Other similar (interval domain-based) analyzers are also not scalable to large code bases [1, 2]. Nevertheless, there have been scalable domain-specific static analyzers, like Astrée [3, 11] and CGS [46], which scale to hundreds of thousands lines of code. However, Astrée targets on programs that do not have recursion and backward gotos, which enables a very efficient interpretation-based analysis [11], and CGS is not fully flow-sensitive [46]. There are other summary-based approaches [16, 17] for scalable global analysis, which are independent of our abstract interpretation-based approach.

8 Conclusion

We have presented a sparse analysis framework. 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 formally present the framework, design two instance analyses, and experimentally show that the sparse versions of the instance analyses far exceeds the performance of non-sparse versions in a realistic setting.

Our results suggest the following guideline in designing sound, 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. By contrast to the common sense in static analysis that scalability is obtained by compromising the analysis precision or soundness, the resulting sparse analysis still preserves the original analysis' precision and soundness.

References

[1] X. Allamigeon, W. Godard, and C. Hymans. Static analysis of string manipulations in critical embedded C programs. In *Proceedings of the International Symposium on Static Analysis*, pages 35–51, 2006.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 binary executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 296–310, 1990.

[6] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, 1991.

[7] J.-D. Choi, V. Sarkar, and E. Schonberg. Incremental computation of static single assignment form. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 223–237, London, UK, UK, 1996. Springer-Verlag.

[8] P. Cousot and R. Cousot. 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*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[10] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astre scale up? *Formal Methods in System Design*, 35(3):229–264, Dec. 2009.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

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

[14] R. K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Trans on Programming Languages and Systems*, 17:487–506, May 1995.

[15] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '92, pages 212–223, New York, NY, USA, 1992. ACM.

[16] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[17] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 187–200, New York, NY, USA, 2011. ACM.

[18] B. Hardekopf and C. Lin. 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, pages 290–299, New York, NY, USA, 2007. ACM.

[19] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, 2009.

[20] B. Hardekopf and C. Lin. 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*, pages 289–298, 2011.

[21] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the International Symposium on Static Analysis*, pages 57–81. Springer-Verlag, 1998.

[22] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *In Computer Aided Verification, CAV'2009*, pages 661–667, 2009.

[23] Y. Jhee, M. Jin, Y. Jung, D. Kim, S. Kong, H. Lee, H. Oh, D. Park, and K. Yi. Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco, `ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf`, January 2008.

[24] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 78–89, 1993.

[25] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 203–217, 2005.

[26] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[27] C. Lattner, A. Lenharth, and V. Adve. 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)*, San Diego, California, June 2007.

[28] W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI 2012: 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2012.

[29] L. Li, C. Cifuentes, and N. Keynes. 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, pages 343–353, New York, NY, USA, 2011. ACM.

[30] J. Lind-Nielson. BuDDy, a binary decision diagram package.

[31] MathWorks. Polyspace embedded software verification. `http://www.mathworks.com/products/polyspace/index.html`.

[32] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *Proceedings of European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[33] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming*, pages 13–25, 2006.

[34] A. Milanova, A. Rountev, and B. G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[35] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM Press, June 2006.

[36] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[37] H. Oh. Large spurious cycle in global static analyses and its algorithmic mitigation. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 14–29, Seoul, Korea, December 2009. Springer-Verlag.

[38] H. Oh, L. Brutschy, and K. Yi. Access analysis-based tight localization of abstract memories. In *VMCAI 2011: 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2011.

[39] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[40] H. Oh and K. Yi. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software: Practice and Experience*, 40(8):585–603, 2010.

[41] H. Oh and K. Yi. Access-based localization with bypassing. In *APLAS 2011: 9th Asian Symposium on Programming Languages and Systems*, volume 7078 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011.

[42] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[43] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–118, 1977.

[44] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 296–309, 2005.

[45] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the International Conference on Compiler Construction*, pages 17–31, 2006.

[46] A. Venet and G. Brat. 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, pages 231–242, New York, NY, USA, 2004. ACM.

[47] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans on Programming Languages and Systems*, 13:181–210, April 1991.

[48] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the International Conference on Computer Aided Verification*, pages 385–398, 2008.

[49] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. 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, pages 218–229, New York, NY, USA, 2010. ACM.

[50] M. Zitser, D. E. S. Group, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *In SIGSOFT 04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

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 10) and uses (Definition 12) for $\hat{F}$, respectively. Let $\hat{F}_a$ be the sparse version of $\hat{F}$ (defined in equation (C)), which is derived using approximated definitions ($\hat{\mathsf{D}}$) and uses ($\hat{\mathsf{U}}$) that satisfy the safety conditions in Definition 21. In this appendix, we prove the Correctness Theorem in Section 2 (Theorem 23):

Theorem 28 (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 29 (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:*

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

□

Note that the notion of helper data dependency $(\sim)$ relaxes the conditions of approximated data dependency $(\rightsquigarrow)$ (Definition 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 it is defined over $(\sim)$ instead of $(\rightsquigarrow)$.

Definition 30 (Helper Abstract Semantic Function)**.**

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

□

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

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

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

Definition 32 (Equivalence of Fixpoint Solutions)**.** *We say* $\mathcal{S} \in \Delta \to \hat{\mathbb{S}}$ *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 33. $\mathbf{lfp}\hat{F} \equiv \mathbf{lfp}\hat{F}_a$.

Proof. We prove the theorem using sub-lemmas that we will prove shortly:

$$\begin{aligned} \mathbf{lfp}\hat{F} &= \mathbf{lfp}\hat{F}_h && \text{(Lemma 34 and Lemma 35)} \\ &\equiv \mathbf{lfp}\hat{F}_a && \text{(Lemma 36)} \end{aligned}$$

□

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

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., $\hat{F}_h(\mathcal{S}) \sqsubseteq \mathcal{S}$:

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

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 $i_0 \hookrightarrow_{\mathcal{S}} \dots \hookrightarrow_{\mathcal{S}} i_n \in \mathsf{Paths}(\mathcal{S})$ 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:

$$\begin{aligned}
\mathcal{S}(i_k)(l) &= \hat{F}(\mathcal{S})(i_k)(l) && (\mathcal{S} = \mathbf{lfp}\hat{F})\\
&= \hat{f}_{i_k}(\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i_k} \mathcal{S}(i'))(l) && \text{(def. of } \hat{F})\\
&= (\bigsqcup_{i' \hookrightarrow_{\mathcal{S}} i_k} \mathcal{S}(i'))(l) && (l \notin \mathsf{D}(i_k) \text{ from } l \notin \hat{\mathsf{D}}(i_k) \text{ and Lemma 11})\\
&\sqsupseteq \mathcal{S}(i_{k-1})(l). && (i_{k-1} \hookrightarrow_{\mathcal{S}} i_k)
\end{aligned}$$

□

Lemma 35. $\mathbf{lfp}\hat{F} \sqsubseteq \mathbf{lfp}\hat{F}_h$.

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., $\hat{F}(\mathcal{S}) \sqsubseteq \mathcal{S}$.

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

It remains to prove the following statement:

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

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 $l \in \hat{\mathbb{L}}$,

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

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 $(\sim)$ (Definition 29), we have $i' \overset{l}{\sim}_{\mathcal{S}} i$. Thus,

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

- $l \notin \hat{\mathsf{D}}(i')$:

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

Here we deduce

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

from $l \notin \mathsf{D}(i')$ and $\mathcal{S} \sqsubseteq \mathbf{lfp}\hat{F}$. Note that, b Lemma 34, $\mathcal{S} = \mathbf{lfp}\hat{F}_h \sqsubseteq \mathbf{lfp}\hat{F}$ holds and we have

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

Then, Lemma 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 36. $\mathbf{lfp}\hat{F}_h \equiv \mathbf{lfp}\hat{F}_a$.

Proof.

$$\begin{aligned}
\mathbf{lfp}\hat{F}_h &= \hat{F}_h(\mathbf{lfp}\hat{F}_a) && (\text{Lemma 37}) \\
&\equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a) && (\text{Lemma 41}) \\
&= \mathbf{lfp}\hat{F}_a
\end{aligned}$$

□

Lemma 37 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 37. $\mathbf{lfp}\hat{F}_h = \hat{F}_h(\mathbf{lfp}\hat{F}_a)$.

Proof.

- $\mathbf{lfp}\hat{F}_h \sqsupseteq \hat{F}_h(\mathbf{lfp}\hat{F}_a)$: By Lemma 31, we have

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

By applying the monotone function $\hat{F}_h$ on both side, 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 $\hat{F}_h$, i.e.,

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

$$\begin{array}{lll} & \mathbf{lfp}\hat{F}_a = \mathbf{lfp}\hat{F}_a & \\ \implies & \hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a) & \text{(Lemma 38, Lemma 31, and Lemma 34)} \\ \implies & \hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \mathbf{lfp}\hat{F}_a & \\ \implies & \hat{F}_h^2(\mathbf{lfp}\hat{F}_a) = \hat{F}_h(\mathbf{lfp}\hat{F}_a). & \text{(Lemma 40)} \end{array}$$

□

Lemma 38 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 38. *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 32.

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 $\hat{F}_h(\hat{\phi})(i)(l) = \hat{F}_a(\hat{\phi}')(i)(l)$:

$$\begin{array}{rll} \hat{F}_h(\hat{\phi})(i)(l) & = \hat{f}_i(\displaystyle\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'})(l) & \text{(def. of } \hat{F}_h\text{)} \\ & = \hat{f}_i((\displaystyle\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'})|_{\hat{\mathsf{U}}(i)})(l) & \text{(Def. 21, Lemma 39, } l \in \hat{\mathsf{D}}(i)\text{, and Def. 15)} \\ & = \hat{f}_i(\displaystyle\bigsqcup_{i' \overset{l'}{\leadsto}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'})(l) & \text{(def. of } \leadsto\text{)} \\ & = \hat{F}_a(\hat{\phi}')(i)(l). & \text{(def. of } \hat{F}_a\text{)} \end{array}$$

   The second equality needs more explanation. Note that Definition 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 39, $\mathbb{U}_{\mathsf{D}(i)}(i) \cup \mathbb{U}_{\hat{\mathsf{D}}(i)\setminus\mathsf{D}(i)}(i) = \mathbb{U}_{\hat{\mathsf{D}}(i)}(i)$ and hence $\mathbb{U}_{\hat{\mathsf{D}}(i)}(i) \subseteq \hat{\mathsf{U}}(i)$ holds. Then, by Definition 15,

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

   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 Definition 6.(3) and 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.
   - $l \notin \hat{\mathsf{D}}(i)$:
   $$\begin{aligned}
   \hat{F}_h(\hat{\phi})(i)(l) &= \hat{f}_i(\bigsqcup_{i' \overset{l'}{\sim}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'})(l) && (\text{def. of } \hat{F}_h)\\
   &= (\bigsqcup_{i' \overset{l'}{\sim}_{\hat{\phi}} i} \hat{\phi}(i')|_{l'})(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \hat{\phi} \sqsubseteq \mathbf{lfp}\hat{F})\\
   &= (\bigsqcup_{i' \overset{l}{\sim}_{\hat{\phi}} i} \hat{\phi}(i')(l))\\
   &= (\bigsqcup_{i' \overset{l}{\sim}_{\hat{\phi}'} i} \hat{\phi}'(i')(l)) && (\hat{\phi} \equiv \hat{\phi}')\\
   &= (\bigsqcup_{i' \overset{l}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')(l)) && (l \in \hat{\mathsf{U}}(i) \text{ and def. of } \rightsquigarrow)\\
   &= (\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'})(l)\\
   &= \hat{f}_i(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{\hat{\phi}'} i} \hat{\phi}'(i')|_{l'})(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \hat{\phi}' \sqsubseteq \mathbf{lfp}\hat{F})\\
   &= \hat{F}_a(\hat{\phi}')(i)(l). && (\text{def. of } \hat{F}_a)
   \end{aligned}$$

□

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

Lemma 39. *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 $l \in \mathbb{U}_{A\cup B}(i)$. 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 $l \in \mathbb{U}_A(i)$. 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 40 shows that the result of applying the helper abstract semantic function $\hat{F}_h$ is invariant up to equivalence.

Lemma 40. *If $\hat{\phi} \equiv \hat{\phi}'$ then $\hat{F}_h(\hat{\phi}) = \hat{F}_h(\hat{\phi}')$.*

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

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

□

Lemma 41. $\hat{F}_h(\mathbf{lfp}\hat{F}_a) \equiv \hat{F}_a(\mathbf{lfp}\hat{F}_a)$.

Proof. By Lemma 38. 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 31 and Lemma 34. □

B Constructing 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 29), it is easy to construct the entire original analysis result $\mathbf{lfp}\hat{F}$ from the sparse analysis result $\mathbf{lfp}\hat{F}_a$, as the following lemma states.

Lemma 42 (Construction)**.** *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 28), we have

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

- $l \notin \hat{\mathsf{D}}(i)$:

$$\begin{aligned}
(\mathbf{lfp}\hat{F})(i)(l) &= (\mathbf{lfp}\hat{F}_h)(i)(l) && \text{(Lemma 34 and Lemma 35)} \\
&= (\hat{F}_h(\mathbf{lfp}\hat{F}_a))(i)(l) && \text{(Lemma 37)} \\
&= \hat{f}_i\Big(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')|_{l'}\Big)(l) && \text{(def. of } \hat{F}_h\text{)} \\
&= \Big(\bigsqcup_{i' \overset{l'}{\rightsquigarrow}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')|_{l'}\Big)(l) && (l \notin \mathsf{D}(i) \text{ from } l \notin \hat{\mathsf{D}}(i) \text{ and } \mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}) \\
&= \bigsqcup_{i' \overset{l}{\rightsquigarrow}_{(\mathbf{lfp}\hat{F}_a)} i} (\mathbf{lfp}\hat{F}_a)(i')(l).
\end{aligned}$$

□

C A Fixpoint Computation Strategy

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

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

We need an efficient algorithm for computing the fixpoint. A naive fixpoint computation strategy, which re-builds data dependency $\rightsquigarrow_{\hat{\phi}}$ whenever the analysis result $\hat{\phi}$ changes, may be inefficient. In practice, we need a smart algorithm that avoids the re-computation 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., [7].

In this appendix, we present a fixpoint computation strategy that reduces re-computation 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 analysis' intermediate results become stable. We slightly modify $\hat{F}_a$ and define the following function:

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

Note that $\hat{F}_l$ uses different arguments in computing analysis results and data dependency: the data dependency $\rightsquigarrow$ 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:

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

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 re-computed as $\hat{\phi}$ is constant in the inner fixpoint iteration. The data dependency is re-computed 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 re-computed only few times. Lemma 43 shows that $\mathbf{lfp}\hat{F}_m$ equals to $\mathbf{lfp}\hat{F}_a$. We first observe two obivous 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 43. $\mathbf{lfp}\hat{F}_m = \mathbf{lfp}\hat{F}_a$.

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

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

- $\mathbf{lfp}\hat{F}_a \sqsubseteq \mathbf{lfp}\hat{F}_m$:

  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,

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

□