Computer Languages, Systems & Structures ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A sparse evaluation technique for detailed semantic analyses ☆

Yoonseok Ko [a], Kihong Heo [b], Hakjoo Oh [b,*]

[a] KAIST, South Korea
[b] Seoul National University, South Korea

ARTICLE INFO

Article history:
Received 4 October 2013
Received in revised form
28 April 2014
Accepted 8 May 2014

Keywords:
Static analysis
Abstract interpretation
Sparse evaluation
Data-flow analysis

ABSTRACT

We present a sparse evaluation technique that is effectively applicable to a set of elaborate semantic-based static analyses. Existing sparse evaluation techniques are effective only when the underlying analyses have comparably low precision. For example, if a pointer analysis precision is not affected by numeric statements like `x:=1` then existing sparse evaluation techniques can remove the statement, but otherwise, the statement cannot be removed. Our technique, which is a fine-grained sparse evaluation technique, is effectively applicable even to elaborate analyses. A key insight of our technique is that, even though a statement is relevant to an analysis, it is typical that analyzing the statement involves only a tiny subset of its input abstract memory and the most are irrelevant. By exploiting this sparsity, our technique transforms the original analysis into a form that does not involve the fine-grained irrelevant semantic behaviors. We formalize our technique within the abstract interpretation framework. In experiments with a C static analyzer, our technique improved the analysis speed by on average 14 ×.

1. Introduction

In static analysis, the technique of sparse evaluation has been widely used to optimize the analysis performance [4,27,11,26,17,14]. Sparse evaluation is based on the observation that static analysis sometimes aggressively abstracts program semantics and therefore a number of program statements are irrelevant to the analysis. For instance, typical pointer analyses (e.g., [14,15]) have comparably low precision and not affected by numeric statements such as `x:=1`. The goal of existing sparse evaluation is to remove such irrelevant statements, which makes the analysis problem smaller and improves the analysis' scalability. In the literature, sparse evaluations have been effectively used to improve the performance of pointer analysis [14,30] and classical data-flow analyses [4,27].

However, existing sparse evaluation techniques are not effective for elaborate semantic analyses in general. Note that the basic assumption of existing sparse evaluation is that the given analysis problem is simple-minded and hence a number of program statements are irrelevant to the analysis. However, in general semantic analyses, it is not uncommon that the analysis is so detailed that the assumption of sparse evaluation does not hold. That is, such an analysis considers all types of values (e.g., including both numbers and pointers) and almost all statements in the program are relevant to the underlying analysis. For instance, consider an elaborate pointer analysis that considers not only pointers but also numeric values.

☆ This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP)/ National Research Foundation of Korea (NRF) (Grant NRF-2008-0062609).

* Corresponding author.
E-mail address: pronto@ropas.snu.ac.kr (H. Oh).

http://dx.doi.org/10.1016/j.cl.2014.05.001
1477-8424/

Then statements like x:=1, which would be irrelevant to simple pointer analyses, are no longer irrelevant and cannot be removed by existing sparse evaluation techniques.

In this paper, we present a new sparse evaluation technique for such detailed static analyses. The intuition behind our technique is that, even though an analysis is detailed and a statement is relevant, the analysis is still sparse in a fine-grained way: analyzing the statement uses only small part of its abstract memory. For example, consider analyzing the statement x:=1. The abstract semantics for the statement would update only the value of x but other values, say y, are not involved in the analysis of the statement. To exploit this sparsity, we first reformulate the analysis problem into an equivalent form that is more fine-grained in that semantic equations are expressed explicitly in terms of individual abstract locations. We call this step decomposition. Then, we define two elimination procedures (which we call no-change and no-contribution eliminations, respectively) that remove the fine-grained irrelevant behaviors of the analysis. We present our technique in the abstract interpretation framework [8,6] and prove that the decomposition and elimination procedures are semantics-preserving, which means that our technique maintains the original analysis' soundness and precision.

Our work provides a general and flexible alternative to the recent sparse analysis framework [23]. Although the goal of ours and the sparse analysis in [23] is the same (i.e., making the analysis sparse), the techniques used are different: the technique in [23] constructs def-use dependencies while we eliminate unnecessary dependencies. This difference makes our technique flexible in controlling the sparseness of the final analysis. We discuss this point in Section 8 in more detail. Furthermore, we generalize the idea of sparse technique in the abstract interpretation framework with arbitrary trace partitioning.

We show the effectiveness of our technique in a realistic setting. We implemented our technique on top of Sparrow, an interval domain-based abstract interpreter [18,16,21,22,24]. In experiments, our technique improved the analysis speed from 2 to 59 times, on average 14 times, and reduced peak memory consumption by 29–80%, on average 56%, for a variety of open-source C benchmarks (6K–111K LOC).

Overview: We illustrate our technique with an example. Suppose that we analyze the program in Fig. 1(a) with a non-relational analysis: the abstract state of the analysis is a map from the set of abstract locations (simply variables x and y in this example) to abstract values, say numeric intervals [8]. During the analysis, the first statement defines the value of x, the second statement defines y, and the last statement updates y with the value of x. Observe that existing sparse evaluation techniques remove no statements in this example, because all the three statements have some effects on the analysis. On the other hand, our technique works as follows:

1. We reformulate the analysis (Fig. 1(a)) into the decomposed form (Fig. 1(b)) in which values of each abstract location are computed separately: each instruction c in Fig. 1(a) is split into (c, x) and (c, y) that hold the values of x and y, respectively. For instance, at node $(1,x)$, the value of x at instruction 1 is stored and value for y at instruction 1 is stored at $(1,y)$. Edge $(c,x)\rightarrow(c',x')$ means that the value of x' at c' may depend on the value of x at c.
2. In Fig. 1(b), we eliminate nodes that have no effect on the analysis, which we call no-change elimination. Note that, among the six nodes in Fig. 1(b), values are actually updated at nodes $(1,x)$, $(2,y)$, and $(3,y)$. At other nodes (dotted ones in Fig. 1(b)), values are not changed. The goal of this step is to remove such "no-change" nodes from the analysis. We simply remove the nodes and short-circuit their in/outflows, which results in Fig. 1(c). Observe that, after no-change elimination, the value of x from $(1,x)$ is directly propagated to $(3,y)$.
3. In Fig. 1(c), we remove irrelevant input flows, which we call no-contribution elimination. In Fig. 1(c), only the value of x is necessary to update the value of y at $(3,y)$; the value of y is not used at $(3,y)$. So, we remove the "no-contribution" flow $(2,y)\rightarrow(3,y)$. Other unnecessary flows are also removed, leading to Fig. 1(d).

With our technique, the original analysis problem (Fig. 1(a)) is reduced to a smaller problem (Fig. 1(d)). Obviously, fixpoint computation for the smaller problem will be cheaper than the original analysis.

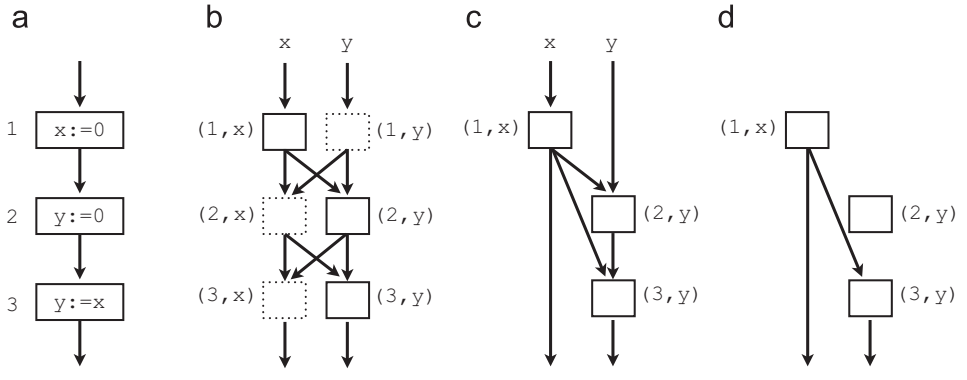

Fig. 1. Example: (a) original analysis, (b) after decomposition, (c) after no-change elimination, and (d) after no-contribution elimination.

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

Outline: Section 2 describes the family of static analyses that this paper considers. Section 3 defines the decomposition step. Section 4 defines two semantic independence involved in the static analysis defined in Section 2. Section 5 formally presents no-contribution and no-elimination procedures. Section 6 shows how to apply our technique to an example analysis. Section 7 presents the experimental results. Section 8 discusses related work. Section 9 concludes the paper.

Notation: In this paper, all functions are considered partial functions. We consider functions that are of type $f \in A \to B$, where A is a set and B is a complete partial order (cpo). We write $\mathrm{dom}(f) \subseteq A$ for the domain of f. we write $f|_C$ for the restriction of function f to the domain $\mathrm{dom}(f) \cap C$. We write $f\backslash_C$ for the restriction of f to the domain $\mathrm{dom}(f) - C$. We abuse the notation $f|_a$ and $f\backslash_a$ for the domain restrictions on singleton set $\{a\}$ and $\mathrm{dom}(f) - \{a\}$, respectively. We write $f[a \mapsto b]$ for the function constructed from function f by changing the value for a to b. We write $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We write $f[\{a_1, \ldots, a_n\} \overset{w}{\mapsto} b]$ for $f[a_1 \mapsto f(a_1) \sqcup b, \ldots, a_n \mapsto f(a_n) \sqcup b]$ (weak update). We write $[a \mapsto b] \in A \to B$ for a function $[a \mapsto b](x) = b$ if $a = x$, otherwise undefined, i.e., $[a \mapsto b](x) = \bot$. For all domains (cpos), we assume an implicit join operator $\sqcup$, ordering relation $\sqsubseteq$, and appropriate $\top$ and $\bot$ elements. In particular, we define $\sqcup, \sqsubseteq, \top, \bot$ for functions in a pointwise fashion, e.g., $f \sqcup g = \lambda x. f(x) \sqcup g(x)$. We write $e.\mathbb{A}$, $e.\mathbb{B}$ to denote the first and the second element of $e \in \mathbb{A} \times \mathbb{B}$, respectively.

2. Abstract interpretation-based static analysis

In this section, we define a class of static analysis that we consider in this paper. We consider transition systems as a program model so that the analysis is general for arbitrary programming language. Thus, our technique is generally applicable to any language.

In Sections 2.1 and 2.2, we define the program and its collecting semantics, respectively. In Section 2.3, we abstract the collecting semantics and derive the abstract semantics that we consider. We suppose basic knowledge of the abstract interpretation framework [8,6] and the trace partitioning [19].

2.1. Programs

We describe a program P as a transition system $(\mathbb{S}, \to, \mathbb{S}_i)$, where $\mathbb{S}$ is the set of states of the program, $(\to) \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation describing how the program execution progresses from one state to the next state, and $\mathbb{S}_i \subseteq \mathbb{S}$ denotes the set of initial states.

We write $\mathbb{S}^\star$ for the set of all finite non-empty sequences of states. When σ is a finite sequence of states, σ_i denotes the $(i+1)$ th state of the sequence, σ_0 the first state, and $\sigma_{\dashv}$ the last state. Given a sequence σ and a state s, $\sigma \cdot s$ denotes a sequence obtained by appending s to σ. A sequence σ is said to be a *trace* of the program P if σ is a partial execution sequence of P, i.e., $\sigma_0 \in \mathbb{S}_i \wedge \forall i. \sigma_i \to \sigma_{i+1}$. We abuse the notation of transition relation $\to$ for traces, i.e., $\sigma' \to \sigma \iff \exists s. \sigma = \sigma' \cdot s \wedge \sigma'_{\dashv} \to s$.

2.2. Collecting semantics

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

$$[\![P]\!] = \{\sigma \in \mathbb{S}^\star \mid \sigma_0 \in \mathbb{S}_i \wedge \forall i. \sigma_i \to \sigma_{i+1}\}$$

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

$$F(\Sigma) \triangleq I \cup \{\sigma \cdot s \mid \sigma \in \Sigma \wedge \sigma_{\dashv} \to s\}$$

where $I = \{s \mid s \in \mathbb{S}_i\}$ is the set of singleton traces of initial states.

2.3. Abstract semantics

We consider the abstract semantics that are obtained from the collecting semantics of the program P by the following Galois connections:

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

The abstract domain $(\Delta \to \hat{\mathbb{S}})$ is obtained by applying the following two abstractions in order.

1. *Partitioning abstraction* (α_1, γ_1): we abstract the set of traces $(\mathcal{P}(\mathbb{S}^\star))$ into partitioned sets of traces $(\Delta \to \mathcal{P}(\mathbb{S}^\star))$, where Δ is the set of partitioning indices, e.g., Δ is the set of control points in the program). Suppose we are given a well-chosen partitioning function $\delta \in \Delta \to \mathcal{P}(\mathbb{S}^\star)$ such that δ is a partition, i.e., $\mathbb{S}^\star = \bigcup_{x \in \Delta} \delta(x)$ and $\forall x, y \in \Delta. x \neq y \Rightarrow \delta(x) \cap \delta(y) = \varnothing$. Then, α_1 and γ_1 are defined as follows: (It is well-known that such α_1 and γ_1 form a Galois-connection [19].)

$$\alpha_1(\Sigma) \triangleq \lambda c \in \Delta. \Sigma \cap \delta(c)$$
$$\gamma_1(\phi) \triangleq \bigcup_{c \in \Delta} \phi(c)$$

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

By choosing a suitable δ, we can define various partitioning strategies [19]. For example, in imperative languages, a state is often decomposed into a control point in $\mathbb{C}$ and a memory state in $\mathbb{M}$, i.e., $\mathbb{S}=\mathbb{C}\times\mathbb{M}$. We use $\mathbb{C}$ as the set of partitioning indices and let $\delta_{\mathbb{C}}\in\mathbb{C}\to\mathcal{P}(\mathbb{S}^{\star})$ partition $\mathbb{S}^{\star}$ based on the final control point: $\delta_{\mathbb{C}}(c)\triangleq\{\sigma\in\mathbb{S}^{\star}\mid\exists m.\sigma_{\dashv}=(c,m)\}$. Then, $\delta_{\mathbb{C}}$ defines the usual flow-sensitive analysis. Likewise, we can define any other trace partitioning such as flow-insensitivity, context-(in)sensitivity, and path-(in)sensitivity by choosing an appropriate δ.

2. *State abstraction* (α_2,γ_2): for each partition, we suppose that the associated set of traces is abstracted into an abstract state $(\hat{\mathbb{S}})$ that over-approximates the reachable states of the traces. α_2 and γ_2 are defined as follows:

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

where $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$ are abstraction and concretization functions for set of traces such that $\mathcal{P}(\mathbb{S}^{\star})\underset{\gamma_{\mathbb{S}}}{\overset{\alpha_{\mathbb{S}}}{\leftrightarrows}}\hat{\mathbb{S}}$. We assume that $\alpha_{\mathbb{S}}$ abstracts the traces in a way that $\hat{\mathbb{S}}$ is a function cpo (complete partial order) $\hat{\mathbb{L}}\to\hat{\mathbb{V}}$ where $\hat{\mathbb{L}}$ is a finite set of abstract locations, and $\hat{\mathbb{V}}$ is an arbitrary cpo for representing abstract values. All non-relational analyses are expressible in this domain. For example, in numerical analysis that uses interval domain, the abstract memory state is a map from all abstract locations to interval values. In addition, the packed relational domain [7,20,31,2] also fits to this domain. Let $Packs\subseteq\mathcal{P}(Var)$ be a set of variable groups (i.e., packs) such that $\bigcup Packs=Var$, and $\mathbb{R}$ be a relational abstract domain such as the octagon domain [20]. Then, the packed relational domain has the form of $Packs\to\mathbb{R}$, which associates each pack with constraints among the variables in the pack. In our framework, we impose that $\hat{\mathbb{L}}$ is fixed prior to the analysis, which means that our technique is not applicable to analyses with, for example, dynamic variable packing.

We consider abstract semantics that is characterized by the least fixpoint of abstract semantic function $\hat{F}\in(\Delta\to\hat{\mathbb{S}})\to(\Delta\to\hat{\mathbb{S}})$ defined as

$$\hat{F}(\hat{X})=\lambda c\in\Delta.\hat{f}_c\left(\bigsqcup_{c'\hookrightarrow c}\hat{X}(c')\right). \tag{1}$$

where $\hat{f}_c\in\hat{\mathbb{S}}\to\hat{\mathbb{S}}$ is the semantic function at partitioning index c and $\hookrightarrow$ is the transition relation between partitioning indices such that

$$(\hookrightarrow)\supseteq\{(c',c)\in\Delta\times\Delta\mid\exists\sigma'\in\delta(c').\sigma'\to\sigma\wedge\sigma\in\delta(c)\}$$

Transition relation $\hookrightarrow$includes all the possible transitions between partitioning indices: $\hookrightarrow$includes a flow (c',c) if there exists a trace σ' in partition c' and the next of σ' belongs to partition c. Note that transition $\to$ is generally unknown prior to the analysis and this is why $\hookrightarrow$ is a superset of the possible transition flows. For example, suppose we analyze an imperative language and we use $\delta_{\mathbb{C}}$ as the partitioning function. Then, $\hookrightarrow$ is identical to the control flow relation of the program, which is often given before the program execution. When analyzing a functional program, for another example, the transition flows can be approximated before the analysis; $\hookrightarrow$ can be given by a separate pre-analysis.

The abstract semantic function $\hat{F}$ should be designed to satisfy $\alpha\circ F\sqsubseteq\hat{F}\circ\alpha$ $(\alpha=\alpha_2\circ\alpha_1)$, then the soundness of the abstract semantics is guaranteed by the fixpoint transfer theorem [9].

3. Pre-processing: decomposition

The first step of our approach is to re-formulate the analysis equation (1) into a fine-grained form, which we call "decomposition". Note that it is not obvious to identify independencies (irrelevant semantic behaviors) directly from the original analysis given in (1), because the semantic function $\hat{f}_c\in\hat{\mathbb{S}}\to\hat{\mathbb{S}}$ works for abstract states but the independencies are usually defined in terms of individual abstract locations. Thus, we first decompose the original analysis so that the analysis is expressed in terms of each abstract location rather than the entire abstract states. In the case of packed relational analyses, the analysis can be expressed in terms of each variable pack rather than each abstract location. With this decomposition, the elimination procedures, which will be defined in Section 5, become simpler.

We first decompose the partitioning indices: Δ is refined by $\hat{\mathbb{L}}$, yielding Δ_d

$$\Delta_d\triangleq\Delta\times\hat{\mathbb{L}}$$

Then, the transition relation $\hookrightarrow$ is decomposed into $\hookrightarrow_d$:

$$\hookrightarrow_d\triangleq\{((c',l'),(c,l))\in\Delta_d\times\Delta_d\mid c'\hookrightarrow c\}$$

That is, each partitioning index (e.g., control point) $c\in\Delta$ is split by the number $(|\hat{\mathbb{L}}|)$ of elements in $\hat{\mathbb{L}}$. One transition $c'\hookrightarrow c$ in the original analysis is represented by $|\hat{\mathbb{L}}|^2$ flows $\{((c',l'),(c,l))\mid l,l'\in\hat{\mathbb{L}}\}$ in the decomposed analysis.

Next, we suppose that each semantic function $\hat{f}_c\in\hat{\mathbb{S}}\to\hat{\mathbb{S}}$ is defined with decomposed functions $\{\hat{f}_{(c,l)}\in\hat{\mathbb{S}}\to\hat{\mathbb{V}}\mid l\in\hat{\mathbb{L}}\}$ such that

$$\hat{f}_c(\hat{s})=\lambda l.\hat{f}_{(c,l)}(\hat{s}) \tag{2}$$

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

In contrast to $\hat{f}_c(\hat{s})$ that computes the entire output state for index c, $\hat{f}_{(c,l)}(\hat{s})$ computes the value for abstract location l only. Other values, say $l' \in \hat{\mathbb{L}}$, are handled by their respective semantic functions, $\hat{f}_{(c,l')}$.

Finally, we define the abstract semantics of decomposed analysis, which is characterized by the least fixpoint of abstract semantic function $\hat{F}_d \in (\Delta_d \to \hat{\mathbb{V}}) \to (\Delta_d \to \hat{\mathbb{V}})$ defined as

$$\hat{F}_d(\hat{X}_d) = \lambda(c,l) \in \Delta_d . \hat{f}_{(c,l)} \left(\bigsqcup_{(c',l') \hookrightarrow_d (c,l)} [l' \mapsto \hat{X}_d(c',l')] \right) \tag{3}$$

The following lemma ensures that the decomposed analysis (3) is essentially the same as the original analysis (1).

Lemma 1. $\forall c \in \Delta,\ l \in \hat{\mathbb{L}}.\ (\mathbf{lfp}\hat{F}_d)(c,l) = (\mathbf{lfp}\hat{F})(c)(l)$.

Proof. We prove the lemma by showing that the semantic function $\hat{F}_d$ is equivalent to $\hat{F}$. Because $\Delta_d = \Delta \times \hat{\mathbb{L}}$, we can change the type of $\hat{F}_d \in (\Delta_d \to \hat{\mathbb{V}}) \to (\Delta_d \to \hat{\mathbb{V}})$ into $\hat{F}_d \in (\Delta \to \hat{\mathbb{L}} \to \hat{\mathbb{V}}) \to (\Delta \to \hat{\mathbb{L}} \to \hat{\mathbb{V}})$ as follows:

$$\hat{F}_d(\hat{X}) = \lambda c \in \Delta . \lambda l \in \hat{\mathbb{L}} . \hat{f}_{(c,l)} \left(\bigsqcup_{(c',l') \hookrightarrow_d (c,l)} [l' \mapsto \hat{X}(c')(l')] \right)$$

Then, we can transform $\hat{F}_d$ into $\hat{F}$ as follows:

$$\begin{aligned}
\hat{F}_d(\hat{X}) &= \lambda c . \lambda l . \hat{f}_{(c,l)} \left(\bigsqcup_{(c',l') \hookrightarrow_d (c,l)} [l' \mapsto \hat{X}(c')(l')] \right) \\
&= \lambda c . \lambda l . \hat{f}_{(c,l)} \left(\bigsqcup_{c' \hookrightarrow c} \bigsqcup_{l' \in \hat{\mathbb{L}}} [l' \mapsto \hat{X}(c')(l')] \right) \quad \cdots \quad \text{def. of } \hookrightarrow_d \\
&= \lambda c . \lambda l . \hat{f}_{(c,l)} \left(\bigsqcup_{c' \hookrightarrow c} \lambda l' . \hat{X}(c')(l') \right) \quad \cdots \quad \text{def. of } \bigsqcup_{\hat{\mathbb{L}} \to \hat{\mathbb{V}}} \\
&= \lambda c . \lambda l . \hat{f}_{(c,l)} \left(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c') \right) \quad \cdots \quad \lambda l' . \hat{X}(c')(l') = \hat{X}(c') \\
&= \lambda c . \hat{f}_c \left(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c') \right) \quad \cdots \quad (2) \qquad \square
\end{aligned}$$

4. Independencies = no-changes or no-contributions

Note that the abstract semantic function given in (3) involves some independencies. For example, when we analyze statement $x := y$, we surely know that the abstract semantic function changes the abstract value of variable x but the function causes *no changes* for other variables. In addition, in order to change the value of x, only the value of y is necessary and other variables have *no contributions* to the change. Nonetheless, the function given in (3) naively follows the transition flows of the program, propagating the entire abstract states from all predecessors to the current partitioning index. In this section, we formally define such independencies (no-changes and no-contributions) involved in the analyses.

4.1. No-changes

We say a decomposed index $(c,l) \in \Delta_d$ is a *no-change* if the semantic function $\hat{f}_{(c,l)}$ does not define new information during the analysis. The most obvious case is when $\hat{f}_{(c,l)}$ has identity transference for all input states, i.e., $\hat{f}_{(c,l)}$ satisfies the condition $\forall \hat{s} \in \hat{\mathbb{S}} . \hat{f}_{(c,l)}(\hat{s}) = \hat{s}(l)$. That is, $\hat{f}_{(c,l)}$ does not modify the value of l but always produces the same value as the value in the input state, $\hat{s}(l)$. However, $\hat{f}_{(c,l)}$ needs not to have identity transference for all inputs; instead, it is sufficient for $\hat{f}_{(c,l)}$ to have identity transference only for those inputs that would occur during the analysis. For example, consider statement $*p := 1$. The semantic function for the statement is not identity for all input states. However, suppose p points to at most location b during the analysis, then, we know that the decomposed semantic functions for other locations $a (\neq b)$ all have identity transference during the analysis because p does not point to other location than b. Thus, in general, (c,l) is a no-change if it satisfies the following condition:

$$\forall \hat{s} \sqsubseteq \mathrm{lfp}_{(c,l)} . \hat{f}_{(c,l)}(\hat{s}) = \hat{s}(l) \tag{4}$$

where $\mathrm{lfp}_{(c,l)}$ denotes the input state to (c,l) at the fixpoint:

$$\mathrm{lfp}_{(c,l)} \triangleq \bigsqcup_{(c',l') \hookrightarrow_d (c,l)} [l' \mapsto (\mathbf{lfp}\hat{F}_d)(c',l')]$$

Note that $\hat{s}$ quantifies over the set of states that possibly occur at index (c,l) during the course of the analysis. Thus, condition (4) means that the semantic function $\hat{f}_{(c,l)}$ does not generate new information during the analysis. We write NChg for the set

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

of all no-change indices, i.e.,

$$\mathsf{NChg} \triangleq \{(c,l) \in \Delta_d \mid \forall \hat{s} \sqsubseteq \mathrm{lfp}_{(c,l)}.\hat{f}_{(c,l)}(\hat{s}) = \hat{s}(l)\}$$

Note that the above definition is mathematical but not finitely computable. At the moment, suppose that we have NChg before the analysis. In Section 5.4, we will show how to approximate NChg so that it is computable in practice.

4.2. No-contributions

We say that an abstract location l' is a *no-contribution* to (c,l) if the value of l' does not contribute to changing the value for (c,l), i.e.,

$$\forall \hat{s} \sqsubseteq \mathrm{lfp}_{(c,l)}.\hat{f}_{(c,l)}(\hat{s}\backslash_{l'}) = \hat{f}_{(c,l)}(\hat{s}) \tag{5}$$

which means that the value of l' is not necessary to produce the value of $\hat{f}_{(c,l)}$ during the course of the analysis. We write $\mathsf{NCon}(c,l)$ for the set of all no-contributions to index (c,l), i.e.,

$$\mathsf{NCon}(c,l) \triangleq \{l' \in \hat{\mathbb{L}} \mid \forall \hat{s} \sqsubseteq \mathrm{lfp}_{(c,l)}.\hat{f}_{(c,l)}(\hat{s}\backslash_{l'}) = \hat{f}_{(c,l)}(\hat{s})\}$$

Note that the above definition is mathematical but not finitely computable. At the moment, suppose that we have NCon before the analysis. In Section 5.4, we will show how to approximate NCon.

5. Removing no-changes and no-contributions

Our goal is to transform the semantic function $\hat{F}_d$ to a sparse version $\hat{F}_s$ that does not involve no-changes and no-contributions. As a result, when we analyze a statement, only the relevant input values are involved in the analysis and only the meaningful values are generated from the statement.

In this section, we show how to eliminate the independencies. By eliminating them, the original analysis $\hat{F}_d$ is transformed into its sparse version $\hat{F}_s$. What are actually transformed is $\hookrightarrow_d$; we transform $\hookrightarrow_d$ into its sparse version $\hookrightarrow_s$ by eliminating no-changes and no-contributions involved in $\hookrightarrow_d$. With $\hookrightarrow_s$, sparse abstract semantic function $\hat{F}_s \in (\Delta_d \to \hat{\mathbb{V}}) \to (\Delta_d \to \hat{\mathbb{V}})$ is defined as follows:

$$\hat{F}_s(\hat{X}_s) = \lambda(c,l) \in \Delta_d.\hat{f}_{(c,l)}\left(\bigsqcup_{(c',l')\hookrightarrow_s(c,l)} [l' \mapsto \hat{X}_s(c',l')]\right) \tag{6}$$

Note that $\hat{F}_s$ is only different from $\hat{F}_d$ in that it is defined over $\hookrightarrow_s$ rather than $\hookrightarrow_d$. Thus, we can reuse abstract semantic function $\hat{f}_{(c,l)}$ and its soundness results from the original analysis design.

We define no-change-elimination and no-contribution-elimination. Because these two transformations remove only the must-independences that do not affect the analysis results, $\hat{F}_s$ should preserve both soundness and precision of original analysis $\hat{F}_d$.

5.1. No-change-elimination

Suppose we are given a no-change index (c, l). Eliminating such a no-change sacrifices neither the analysis' correctness nor precision because, no-change indices are just placeholders that do not generate new information during the analysis.

When (c, l) is a no-change, we remove (c, l) and rearrange the remaining flow edges properly. For example, consider Fig. 2: the leftmost (respectively, rightmost) figure represents the flow relation ($\hookrightarrow_d$) before (respectively, after) applying no-change-elimination to (c, l). In Fig. 2, suppose (c, l) is a no-change point. Then, we observe its two implications: (1) the abstract value associated with (p, l), a predecessor of (c, l), is the only one that is needed to generate the value for (c, l) because the semantic function $\hat{f}_{(c,l)}$ simply propagates the value of (p, l) without any change (in other words, other values

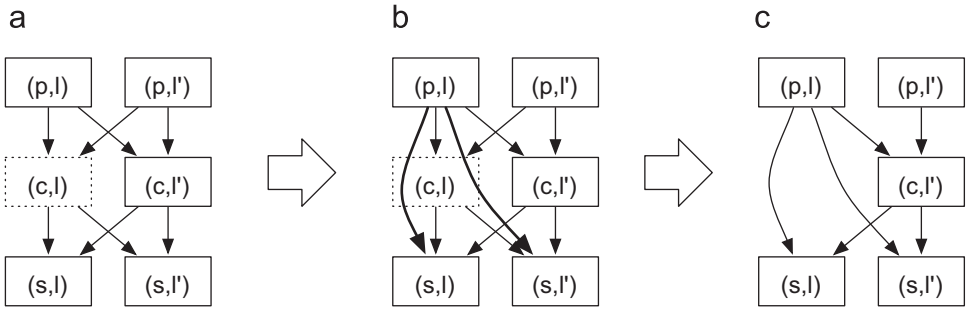

Fig. 2. No-change elimination. If the value at (c, l) is guaranteed to have no changes during the analysis, bypassing (c, l) is safe and preserves the analysis precision.

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

such as (p,l') are not necessary); (2) in order for the value of (p,l) to be propagated to (s,l) and (s,l'), the value needs not pass through (c,l) because the value is not changed by $\hat{f}_{(c,l)}$. From these observations, we directly connect the input flow $((p,l),(c,l))$ into all the successors of (c,l), adding the thick flows in Fig. 2(b). Then, we remove (c,l) as well as flow edges connected to (c,l), leading to Fig. 2(c).

Formally, given a no-change point (c,l), the no-change-elimination is defined by transformation $\mathcal{T}^{(c,l)}_{noChg} \in \mathcal{P}(\Delta_d \times \Delta_d) \to \mathcal{P}(\Delta_d \times \Delta_d)$ defined as follows:

$$\begin{aligned}\mathcal{T}^{(c,l)}_{noChg}(\hookrightarrow) \triangleq (\hookrightarrow) \cup \{((p,l),(s,l'))|(p,l)\hookrightarrow(c,l) \wedge (c,l)\hookrightarrow(s,l')\} \quad &\cdots \quad ①\\ \setminus\{((p,l'),(c,l))|(p,l')\hookrightarrow(c,l)\} \quad &\cdots \quad ②\\ \setminus\{((c,l),(s,l'))|(c,l)\hookrightarrow(s,l')\} \quad &\cdots \quad ③\end{aligned}$$

Given a no-change point (c,l), $\mathcal{T}^{(c,l)}_{noChg}$ performs three operations: ① it connects (p,l) to all of the successors of (c,l); ② it removes all the inflows $((p,l'),(c,l))$; and ③ it removes all the outflows $((c,l),(s,l'))$. In effect, after the elimination, the no-change point (c,l) is ignored during the analysis. The following lemma ensures the correctness of $\mathcal{T}^{(c,l)}_{noChg}$.

Lemma 2. *Let $(c,l) \in \Delta_d$ be a no-change point. Let $\hookrightarrow_d$ and $\hookrightarrow'_d$ be flow relations such that $(\hookrightarrow'_d) = \mathcal{T}^{(c,l)}_{noChg}(\hookrightarrow_d)$. Let $\mathcal{S}'$ and $\mathcal{S}$ be $\mathbf{lfp}\hat{F}_d{}'$ and $\mathbf{lfp}\hat{F}_d$, respectively, where*

$$\hat{F}_d{}'(\hat{X}) = \lambda(c,l) \in \Delta_d.\hat{f}_{(c,l)}\left(\bigsqcup_{(c',l')\hookrightarrow'_d(c,l)} [l' \mapsto \hat{X}(c',l')]\right)$$

$$\hat{F}_d(\hat{X}) = \lambda(c,l) \in \Delta_d.\hat{f}_{(c,l)}\left(\bigsqcup_{(c',l')\hookrightarrow_d(c,l)} [l' \mapsto \hat{X}(c',l')]\right).$$

Then,

$$\forall (c',l') \in \Delta_d \quad \text{s.t. } (c',l') \neq (c,l).\mathcal{S}'(c',l') = \mathcal{S}(c',l').$$

Proof. We prove the lemma by showing that the fixpoint equation of $\hat{F}_d{}'$ is equivalent to the equation of $\hat{F}_d$ up to the points that remain after the transformation. For simplicity, consider the case with the following assumptions: $\hat{\mathbb{L}} = \{l\}$ and $\hookrightarrow_d = \{((p,l),(c,l)),((c,l),(s,l))\}$ (it is easy to extend this proof to the general case.). Then, the fixpoint equations of $\hat{F}_d$ are as follows:

$$\begin{aligned}\mathcal{S}(c,l) &= \hat{f}_{(c,l)}([l \mapsto \mathcal{S}(p,l)])\\ \mathcal{S}(s,l) &= \hat{f}_{(s,l)}([l \mapsto \mathcal{S}(c,l)])\end{aligned} \tag{7}$$

We can transform the equation into the fixpoint equation of $\hat{F}_d{}'$ as follows:

$$\begin{aligned}\mathcal{S}(s,l) &= \hat{f}_{(s,l)}([l \mapsto \mathcal{S}(c,l)]) \quad \cdots \quad \text{by (7)}\\ &= \hat{f}_{(s,l)}([l \mapsto \hat{f}_{(c,l)}([l \mapsto \mathcal{S}(p,l)])]) \quad \cdots \quad \text{by (7)}\\ &= \hat{f}_{(s,l)}([l \mapsto \mathcal{S}(p,l)]) \quad \cdots \quad \hat{f}_{(c,l)}(\hat{s}) = \hat{s}(l)\end{aligned}$$

By the definition of $\mathcal{T}^{(c,l)}_{noChg}$, $\hookrightarrow'_d$ is $\{((p,l),(s,l))\}$. Thus, the fixpoint equation of $\hat{F}_d{}'$ is $\mathcal{S}' = \hat{f}_{(s,l)}([l \mapsto \mathcal{S}(p,l)])$, which is equivalent to the equations of $\hat{F}_d$ as derived above. □

5.2. No-contribution-elimination

The second one removes no-contributions from the analysis. It is enough to apply the no-contribution elimination only to the change points where values are ever changed during the analysis, because independencies involved in no-changes are all removed by the no-change elimination.

Suppose we are given a flow $((p,l'),(c,l)) \in (\hookrightarrow_d)$ to a change point. We say the flow is no-contribution flow if $l' \in \mathsf{NCon}(c,l)$. The goal of the no-contribution-elimination is to remove all such flows, which is also safe because no-contribution flows are, by definition, unnecessary in generating new information during the course of the analysis.

For each no-contribution flow $((p,l'),(c,l))$, we simply remove it from $\hookrightarrow_d$. For example, consider Fig. 3: the left (respectively, right) figure represents the flow relation $(\hookrightarrow_d)$ before (respectively, after) applying no-contribution-elimination to $((p,l'),(c,l))$. In Fig. 3, suppose (c,l) is a change point, i.e., $\exists \hat{s} \sqsubseteq \mathrm{lfp}_{(c,l)}.\hat{f}_{(c,l)}(\hat{s}) \neq \hat{s}(l)$, and $((p,l'),(c,l))$ is a no-contribution flow. In this case, we can safely remove the flow because it is unnecessary to generate the value of (c,l).

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

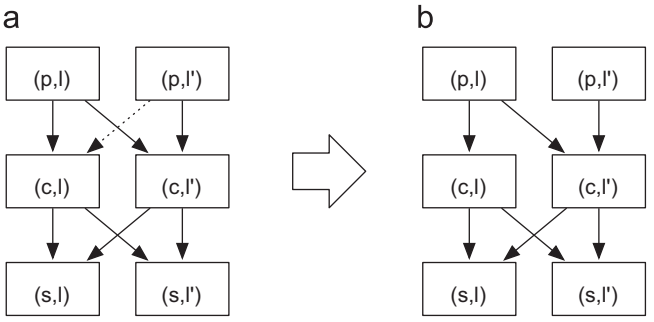

Fig. 3. No-contribution elimination. If flow $((p,l'),(c,l))$ is guaranteed to have no contributions to generating the values at (c,l), removing the value flow is still safe and precision-preserving.

Formally, given a no-contribution flow $((p,l'),(c,l))$, the no-contribution-elimination is simply defined by transformation $\mathcal{T}_{noCon}^{((p,l'),(c,l))}$ as follows:

$$\mathcal{T}_{noCon}^{((p,l'),(c,l))}(\hookrightarrow) \triangleq (\hookrightarrow)\backslash\{((p,l'),(c,l))\}.$$

Lemma 3. *Let $((p,l'),(c,l))$ be a no-contribution flow. Let $\hookrightarrow_d$ and $\hookrightarrow_d'$ be flow relations such that $(\hookrightarrow_d') = \mathcal{T}_{noCon}^{((p,l'),(c,l))}(\hookrightarrow_d)$. Let $\mathcal{S}'$ and $\mathcal{S}$ be $\mathbf{lfp}\hat{F}_d{}'$ and $\mathbf{lfp}\hat{F}_d$, respectively, where*

$$\hat{F}_d{}'(\hat{X}) = \lambda(c,l)\in\Delta_d.\hat{f}_{(c,l)}\left(\bigsqcup_{(c',l')\hookrightarrow_d'(c,l)}[l'\mapsto\hat{X}(c',l')]\right)$$

$$\hat{F}_d(\hat{X}) = \lambda(c,l)\in\Delta_d.\hat{f}_{(c,l)}\left(\bigsqcup_{(c',l')\hookrightarrow_d(c,l)}[l'\mapsto\hat{X}(c',l')]\right)$$

Then,

$$\forall(c,l)\in\Delta_d.\mathcal{S}'(c,l) = \mathcal{S}(c,l).$$

Proof. We prove the lemma by showing that the fixpoint equation of $\hat{F}_d{}'$ is equivalent to the equation of $\hat{F}_d$. For simplicity, consider the case with the following assumptions: $\hookrightarrow_d = \{((p,l),(c,l)),((p,l'),(c,l))\}$ and $l'\in\mathsf{NCon}(c,l)$ (it is easy to extend this proof to the general case.). Then, the fixpoint equations of $\hat{F}_d$ are as follows:

$$\mathcal{S}(c,l) = \hat{f}_{(c,l)}([l\mapsto\mathcal{S}(p,l)]\sqcup[l'\mapsto\mathcal{S}(p,l')]) \tag{8}$$

We can transform the equation into the fixpoint equation of $\hat{F}_d{}'$ as follows:

$$\begin{aligned}\mathcal{S}(c,l) &= \hat{f}_{(c,l)}((l\mapsto\mathcal{S}(p,l))\sqcup(l'\mapsto\mathcal{S}(p,l')))\\ &= \hat{f}_{(c,l)}([l\mapsto\mathcal{S}(p,l),l'\mapsto\mathcal{S}(p,l')]) \quad\cdots\quad \text{def. of } \sqcup\\ &= \hat{f}_{(c,l)}([l\mapsto\mathcal{S}(p,l),l'\mapsto\mathcal{S}(p,l')]\backslash_{l'}) \quad\cdots\quad l'\in\mathsf{NCon}(c,l)\\ &= \hat{f}_{(c,l)}([l\mapsto\mathcal{S}(p,l)]) \quad\cdots\quad \text{def. of } \backslash\end{aligned}$$

By the definition of $\mathcal{T}_{noCon}^{((p,l'),(c,l))}$, $\hookrightarrow_d' = \{((p,l),(c,l))\}$. Thus, the fixpoint equation of $\hat{F}_d{}'$ is $\mathcal{S}' = \hat{f}_{(c,l)}([l\mapsto\mathcal{S}(c,l)])$, which is equivalent to the equations of $\hat{F}_d$ as derived above. □

5.3. Final analysis

The final flow relation $\hookrightarrow_s$ is obtained by applying the two eliminations to $\hookrightarrow_d$ until stabilized. No-change-elimination and no-contribution-elimination do not interfere with each other because no-change-elimination is applied only to no-change points and the no-contribution-elimination is applied only to change points. Thus, we can repeatedly apply the transformations in any order. Both transformations eventually terminate because the space Δ_d is finite. With $\hookrightarrow_s$, we obtain the final sparse semantic function (6).

Note that we do not need to apply the transformations until stabilized. Because the analysis semantics is preserved over the elementary step of both transformations ($\mathcal{T}_{noChg}^{(c,l)}$ and $\mathcal{T}_{noCon}^{((p,l'),(c,l))}$), we can stop the elimination procedures at any desired point.

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

5.4. Approximating no-changes and no-contributions

In practice, we need to approximate NChg and NCon. Our method presented so far serves as a purpose of defining a mathematically correct analysis; it does not yet serve for practical uses. The main reason is that NChg and NCon are defined with the original fixpoint $\mathbf{lfp}\hat{F}$ computed; thus, in order to compute NChg and NCon, we need to first compute $\mathbf{lfp}\hat{F}$, the analysis that we actually want to make sparse. Therefore, we should use approximations of NChg and NCon, which, from now on, we denote $\widehat{\mathsf{NChg}}$ and $\widehat{\mathsf{NCon}}$, respectively. Then, no-change-elimination and no-contribution-elimination are also approximated in that they operate with $\widehat{\mathsf{NChg}}$ and $\widehat{\mathsf{NCon}}$ instead of NChg and NCon.

For correctness, $\widehat{\mathsf{NChg}}$ and $\widehat{\mathsf{NCon}}$ should be under-approximations of NChg and NCon, respectively. This condition is intuitive and we can easily find a counterexample if the condition is not satisfied. Suppose $(c,l)\in\widehat{\mathsf{NChg}}$ and $(c,l)\notin\mathsf{NChg}$ for abstract location (c,l). Performing the no-change-elimination with $\widehat{\mathsf{NChg}}$ is unsafe because (c,l) is not an actual no-change point ($(c,l)\notin\mathsf{NChg}$).

It is easy to check that the analysis obtained by removing approximated independencies are still correct and precision-preserving. We already proved that eliminating no-changes and no-contributions is safe. The approximated independencies are subsets of concrete ones, i.e., $\widehat{\mathsf{NChg}}\subseteq\mathsf{NChg}$ and $\forall(c,l)\in\Delta_d.\widehat{\mathsf{NCon}}(c,l)\subseteq\mathsf{NCon}(c,l)$, and therefore they are clearly no-changes and no-contributions and removing them is safe. However, because of the approximation, some no-changes and no-contributions would be still present in the final analysis.

6. An instantiation example: a non-relational analysis

In this section, we show how our technique is applied to a non-relational static analysis. In Section 6.1, we describe a framework for a simple class of non-relational abstract interpretation, and in Section 6.2, we show how to find $\widehat{\mathsf{NChg}}$ and $\widehat{\mathsf{NCon}}$ for the analysis.

6.1. Baseline analysis

Language: We assume that the partitioning indices Δ are control points of the program and each partitioning index $c\in\Delta$ is associated with a command. We consider a simple imperative language that has integers and pointers as values. The syntax of the language is as follows:

$$cmd\to \mathsf{assign}(x,e)|\mathsf{assign}(*x,e)|\mathsf{assume}(x<n)$$
$$e\to n|x|\&x|*x|e+e$$

Command $\mathsf{assign}(x,e)$ assigns the value of expression e to variable x. $\mathsf{assign}(*x,e)$ performs indirect assignments; the value of e is assigned to the location that x points to. $\mathsf{assume}(x<n)$ makes the program continue only when the condition is satisfied.

Abstract semantics: Note that our framework requires abstract states to be maps from abstract locations to values (Section 2.3). Consider the following abstract domains:

$$\hat{\mathbb{S}}=\hat{\mathbb{L}}\to\hat{\mathbb{V}},\quad \hat{\mathbb{L}}=Var,\quad \hat{\mathbb{V}}=\hat{\mathbb{Z}}\times\hat{\mathbb{P}},\quad \hat{\mathbb{P}}=\mathcal{P}(\hat{\mathbb{L}})$$

An abstract state ($\hat{\mathbb{S}}$) is a map from a set of abstract locations ($\hat{\mathbb{L}}$) to a domain of abstract values ($\hat{\mathbb{V}}$). We treat each program variable as an abstract location. An abstract value is a pair of an abstract integer ($\hat{\mathbb{Z}}$ such that $\mathcal{P}(\mathbb{Z})\underset{\gamma_{\mathbb{Z}}}{\overset{\alpha_{\mathbb{Z}}}{\leftrightarrows}}\hat{\mathbb{Z}}$) and an abstract pointer ($\hat{\mathbb{P}}$). An abstract pointer is a set of abstract locations. Note that the domain is generic so we can freely choose any non-relational numeric domains for $\hat{\mathbb{Z}}$, such as intervals [8].

The abstract semantics is defined by the least fixpoint of semantic function $\hat{F}$ as described in Eq. (1), where the abstract semantic function $\hat{f}_c\in\hat{\mathbb{S}}\to\hat{\mathbb{S}}$ is defined as follows: (*Notation*: we write $e.\mathbb{A}$, $e.\mathbb{B}$ to denote the first and the second element of $e\in\mathbb{A}\times\mathbb{B}$, respectively.)

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

The abstract effect of $\mathsf{assign}(x,e)$ is to update the value of x with the evaluation result of e. $\mathsf{assign}(*x,e)$ weakly updates the value of abstract locations that $*x$ denotes.[1] $\mathsf{assume}(x<n)$ prunes the numeric value of x according to the condition. The abstract value of expression e is computed by an auxiliary function $\hat{\mathcal{E}}\in e\to\hat{\mathbb{S}}\to\hat{\mathbb{V}}$, defined as follows:

$$\hat{\mathcal{E}}(n)(\hat{s})=\langle\alpha_{\mathbb{Z}}(\{n\}),\bot\rangle$$
$$\hat{\mathcal{E}}(x)(\hat{s})=\hat{s}(x)$$
$$\hat{\mathcal{E}}(\&x)(\hat{s})=\langle\bot,\{x\}\rangle$$

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

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

$$\hat{\mathcal{E}}(*x)(\hat{s}) = \bigsqcup\{\hat{s}(a) | a \in \hat{s}(x).\hat{\mathbb{P}}\}$$
$$\hat{\mathcal{E}}(e_1+e_2)(\hat{s}) = \langle v_1.\hat{\mathbb{Z}} \hat{+}_{\hat{\mathbb{Z}}} v_2.\hat{\mathbb{Z}}, v_1.\hat{\mathbb{P}} \cup v_2.\hat{\mathbb{P}} \rangle$$
$$\text{where } v_1 = \hat{\mathcal{E}}(e_1)(\hat{s}), \quad v_2 = \hat{\mathcal{E}}(e_2)(\hat{s})$$

6.2. Finding no-changes and no-contributions

To approximate $\widehat{\mathsf{NChg}}$ and $\widehat{\mathsf{NCon}}$, we use a conservative pre-analysis. The pre-analysis we use is a flow-insensitive version of the baseline analysis:

$$\Delta \to \hat{\mathbb{S}} \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} \hat{\mathbb{S}} \quad \alpha_p = \lambda \hat{X}.\bigsqcup\{\hat{X}(c) | c \in \mathsf{dom}(\hat{X})\} \quad \hat{F}_p = \lambda\, \hat{s}.\, \bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s})$$

Other than the above pre-analysis, any sound approximation of the original analysis can be used as a pre-analysis.

Let $\hat{\mathcal{T}}_p \in \hat{\mathbb{S}}$ be the pre-analysis results. Using $\hat{\mathcal{T}}_p$, we approximate no-changes and no-contributions in the analysis. For each partitioning index c, we compute $\widehat{\mathsf{NChg}}$ by excluding abstract locations whose values are possibly modified at c

$$\widehat{\mathsf{NChg}} = \{(c,l) \in \Delta \times \hat{\mathbb{L}} | l \in (\hat{\mathbb{L}} - \{x\}) \wedge (c = \mathsf{assign}(x,e) \text{ or } \mathsf{assume}(x < n))\}$$
$$\cup \{(c,l) \in \Delta \times \hat{\mathbb{L}} | l \in (\hat{\mathbb{L}} - \hat{\mathcal{T}}_p(x).\hat{\mathbb{P}}) \wedge c = \mathsf{assign}(*x,e)\}$$

According to the definition of $\hat{f}_{(c,l)}$, $\mathsf{assign}(x,e)$ and $\mathsf{assume}(x<n)$ do not define any abstract locations except x. $\mathsf{assign}(*x,e)$ may change values of abstract locations in $\hat{\mathcal{T}}_p(x).\hat{\mathbb{P}}$. Because the pre-analysis is conservative, we ensure that all abstract locations in $\hat{\mathbb{L}} - \hat{\mathcal{T}}_p(x).\hat{\mathbb{P}}$ are never modified during the actual analysis.

We compute $\widehat{\mathsf{NCon}}(c,l)$ by excluding abstract locations that are possibly used when analyzing control point c:

$$\widehat{\mathsf{NCon}}(c,l) = \begin{cases} \hat{\mathbb{L}} - \mathcal{U}(e)(\hat{\mathcal{T}}_p) & c = \mathsf{assign}(x,e) \\ \hat{\mathbb{L}} - (\{x\} \cup \hat{\mathcal{T}}_p(x).\hat{\mathbb{P}} \cup \mathcal{U}(e)(\hat{\mathcal{T}}_p(c))) & c = \mathsf{assign}(*x,e) \\ \hat{\mathbb{L}} - \{x\} & c = \mathsf{assume}(x<n) \end{cases}$$

where $\mathcal{U}(e)(\hat{s})$ computes the set of abstract locations that are used during the evaluation of $\hat{\mathcal{E}}(e)(\hat{s})$:

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

The definition of $\widehat{\mathsf{NCon}}(c,l)$ is also naturally derived from the definition of $\hat{f}_c$. When $\mathsf{cmd}(c)$ is $\mathsf{assign}(x,e)$, all locations except those in $\mathcal{U}(e)(\hat{\mathcal{T}}_p)$ are unnecessary. Analyzing $\mathsf{assign}(*x,e)$ requires location x (because of semantic definition of weak updates), the points-to set of x ($\hat{\mathcal{T}}_p(x).\hat{\mathbb{P}}$), and $\mathcal{U}(e)(\hat{\mathcal{T}}_p)$. Note that in the above definition, for simplicity, we aggressively approximated $\mathsf{NCon}(c,l)$ even without considering the current location l. If this is unsatisfactory, we can design $\mathsf{NCon}(c,l)$ more precisely, discriminating each l, which increases the sparseness of the final sparse analysis.

7. Experiments

In this section, we check the performance of our technique in practice. We implemented our technique on top of our Sparrow analysis framework [18,16,21,22,24]. The analyzer is for detecting memory safety violations such as buffer overruns in C programs, based on flow-sensitive abstract interpretation with the interval domain [8]. Essentially, the abstract domain of the analyzer is the same with that presented in the previous section, but we use intervals for the value domain ($\hat{\mathbb{V}}$).

Setting: We used 15 open source softwares as benchmarks (Table 1). The benchmarks are various open-source applications, and most of them are from GNU open-source projects. Standard library calls are summarized using handcrafted function stubs. For other unknown procedure calls to external code, we assume 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.

The baseline analyzer, Baseline, is a version of Sparrow that performs the access-based localization. With the access-based localization technique [22], each code block is analyzed with only the to-be-accessed parts of its input state, reducing a lot of analysis time and memory consumption. It has been reported that the technique is faster by up to 50 times than the conventional, reachability-based localization technique [25]. More details on the baseline analysis are available at [25].

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

Table 1
Experimental results. **LOC** reports lines of code obtained by running `wc` on the source codes before preprocessing. **Time** and **Mem** are analysis time (in s) and peak memory consumption (in MB) of the various versions of analyses. **Time** for Sparse includes the time for both sparsification and actual analysis time. ∞ means the analysis ran out of time (exceeded 24 h time limit) or out of memory (exceeded 24 GB). **Spd↑** is the speed-up of Sparse over Baseline. **Mem↓** shows the memory savings.

Programs	LOC	Baseline		Sparse		Spd↑	Mem↓
		Time	Mem	Time	Mem		
httptunnel-3.3	6K	15	59	7	42	2×	29%
gzip-1.2.4a	7K	68	133	16	69	4×	48%
parser	10K	832	338	99	181	8×	47%
twolf	19K	439	370	71	169	6×	54%
sed-4.0.8	25K	249	303	108	180	2×	41%
tar-1.13	20K	7489	2133	431	216	4×	64%
make-3.76.1	27K	1830	1080	1148	560	7×	48%
wget-1.9	35K	3049	934	442	369	7×	60%
screen-4.0.2	45K	∞	∞	9915	1758	N/A	N/A
bison-2.4	56K	10,541	180	180	237	59×	58%
lighttpd-1.4.24	57K	8983	326	326	366	28×	57%
a2ps-4.14	64K	12,831	671	672	476	19×	74%
gsasl-1.6.0	91K	∞	∞	6742	372	N/A	N/A
bash-2.05a	105K	1029	684	120	188	9×	73%
lsh-2.0.4	111K	30,456	1660	1407	333	22×	80%

We applied our technique to Baseline and derived Sparse. In implementing Sparse, we do not need to explicitly perform the decomposition step. Instead, we distinguish decomposed control points implicitly by tagging every edge with a relevant set of abstract locations between the control points and manipulate the edges. When we remove a no-change $(c, l) \in \widehat{\mathsf{NChg}}$, we draw a new edge between a predecessor and a successor of c, tag the edge with $\{l\}$, and remove l from the original set. Likewise, when we remove a no-contribution $l' \in \widehat{\mathsf{NCon}}(c, l)$, we just remove l' from the set on the incoming edge of c. We applied our technique intraprocedurally: only the nodes inside procedures are made sparse and interprocedural edges remain unchanged.

Results: Table 1 shows the impact of our technique. We compare analysis time and peak memory consumption of the two analyzers. Because all of the analyzers share the same front-end, we only measure analysis time.

Our technique definitely improves the analysis performance in terms of both time and memory. Sparse is faster by 2–59 times and saves memory consumption by 29–80% compared to Baseline. Furthermore, Sparse is able to analyze large and complex programs. In our experiments, the most complex program in the benchmarks, screen-4.0.2, can be analyzed within 3 h and 1.8 GB of memory.

Sparse outperforms Baseline because Sparse more frequently localizes abstract memories and skips meaningless propagations. In our experiments, 97.4–99.9% of abstract locations were no-changes or no-contributions at each control point. Moreover, the improvement is more outstanding in analyzing large programs: 2–7× speed-up and 29–60% of memory reduction for small programs (less than 45K LOC) but 9–59× speed-up and 58–80% of memory reduction for the larger programs (more than 45K LOC).

8. Related work and discussion

Our key contribution, and the main difference from the existing sparse evaluation techniques [4,27], is that we combine the "decomposition" and "sparsification" procedures together within the abstract interpretation framework. Although the basic ideas of those two procedures (decomposition and sparsification) are not entirely new individually, they have not been used together in the previous literature.

Previous sparse evaluation techniques [4,27] are primarily concerned with the sparsification process only. Thus, existing approaches are not effective "as is" for semantic analyses that we consider in this paper. In other words, previous sparse evaluation techniques are effective only when the input analysis problem is already defined in a sparse form to which the sparsification process is directly applicable. On the other hand, our technique employs a decomposition step and make sparsification effective even when the original analysis is not favorable for the sparsification process.

The program dependence graph (PDG) [13] also does not utilize the decomposition step and hence not suitable for detailed semantic analyses. PDGs are an intermediate representation that makes explicit the data and control dependences of the program. However, as in the previous sparse evaluation techniques, PDGs do not provide fine-grained dependences that are essential for sparsifying the class of detailed semantic analyses, as described in this paper.

The decomposition idea has also been used previously but for different purposes. For example, in [29], the domain of abstract environment, which is a map from abstract locations to abstract values, is decomposed into the "exploded graph" representation, which is similar to our decomposition step. In this paper, we put the decomposition and sparsification ideas

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

together within the abstract interpretation framework and show that the sparse evaluation idea can be used more broadly than before.

In the literature, there exist two approaches to make an analysis sparse. One approach is *sparse evaluation* [4,27] and the other is *sparse analysis* [28,32,12,3,14,15,23]. While conventional static analyses propagate abstract values along control flows of the program, sparse analysis techniques use def-use dependencies (often in the SSA form) of the program and directly propagate abstract values from their definition points to their use points.

Both approaches have their own merits and hence can be complementarily used in practice. For example, the sparse analysis approach may be more convenient to implement than the sparse evaluation approach, because there exist many practical algorithms for dependency generation and manipulation (such as SSA algorithms [10,1,5]) and therefore the sparsifying process can be effectively realized on top of the off-the-shelf implementations [14,15]. On the other hand, the sparse evaluation approach is more flexible in controlling the analysis' sparseness than the sparse analysis approach. Note that, with the sparse analysis method, a safe sparse analysis is obtained only after the sparsifying process is completely finished because the method gathers *may-dependencies* and all the dependencies are required for soundness. By contrast, in a sparse evaluation approach, it is always safe to stop the sparsifying process at any desired point during the analysis because the approach eliminates only the *must-independencies*. This flexibility would be useful in practice when tuning the balance of costs between sparsifying process and actual analysis.

Given that sparse evaluation and sparse analysis approaches may have different merits in practice, it would be good if analysis designers can freely choose the most appropriate method depending on their situations, or combine the merits of both approaches. Unfortunately, analysis designers could not use the sparse evaluation approach when developing detailed static analyses. Our technique can be considered as an effort to give analysis designers more flexibility when choosing sparse analysis techniques.

Our work improves the access-based localization technique [22] in two ways. The access-based localization reduces the cost of interprocedural analysis by analyzing procedures with only the portion of the abstract states that will be accessed by the procedures. Similar to our technique, the access information is computed by a flow-insensitive pre-analysis. Theoretically, our no-contribution elimination (Section 5.2) can be understood as a fine-grained version of the localization technique: while the previous technique [22] applies localization only at procedure entries, we localize abstract states at every statement (including procedure entries). Furthermore, our technique also applies the no-change elimination (Section 5.1), which is entirely new compared to the access-based localization technique.

9. Conclusion

In this paper, we have presented a new sparse evaluation technique and showed that the sparse evaluation ideas can be applicable even to elaborate static analyses in general. Compared to existing sparse evaluation techniques, our technique introduces a new preprocessing step (called “decomposition”) that reformulates the original analysis into a form where semantic irrelevance is explicitly represented. After the decomposition, our technique makes analysis sparse by eliminating semantic independencies (called “no-changes” and “no-contributions”) involved in the usual abstract interpretation. The elimination process is semantics-preserving in that both correctness and precision of the analysis are preserved. We show the effectiveness of the technique by demonstrating the achieved speed-up of a realistic C global analyzer with open-source C benchmarks.

References

[1] Aycock John, Horspool R Nigel. Simple generation of static single-assignment form. In: Proceedings of the 9th international conference on compiler construction, CC '00. London, UK: Springer-Verlag; 2000. p. 110–24.
[2] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, et al. A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN-SIGACT conference on programming language design and implementation; 2003. p. 196–207.
[3] Chase David R, Wegman Mark, Zadeck F Kenneth. Analysis of pointers and structures, In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation; 1990. p. 296–310.
[4] Choi Jong-Deok, Cytron Ron, Ferrante Jeanne. Automatic construction of sparse data flow evaluation graphs. In: Proceedings of the ACM SIGPLAN-SIGACT symposium on principles of programming languages; 1991. p. 55–66.
[5] Chow Fred C, Chan Sun, Liu Shin-Ming, Lo Raymond, Streich Mark. Effective representation of aliases and indirect memory operations in SSA form. In: Proceedings of the 6th international conference on compiler construction, CC '96. London, UK: Springer-Verlag; 1996. p. 253–67.
[6] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Conference record of the sixth annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. San Antonio, TX, New York, NY: ACM Press; 1979. p. 269–82.
[7] Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Rival X. Why does astrée scale up? Form Methods Syst Des 2009;35(December (3)):229–64.
[8] Cousot Patrick, Cousot Radhia. 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; 1977. p. 238–52.
[9] Cousot Patrick, Cousot Radhia. Abstract interpretation frameworks. J Logic Comput 1992;2(4):511–47.
[10] Cytron Ron, Ferrante Jeanne, Rosen Barry K, Wegman Mark N, Zadeck F Kenneth. Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst 1991;13(October (4)):451–90.
[11] Cytron Ron K, Ferrante Jeanne. Efficiently computing ϕ-nodes on-the-fly. ACM Trans Program Lang Syst 1995;17(May):487–506.
[12] Dhamdhere M Dhananjay, Rosen Barry K, Zadeck F Kenneth. How to analyze large programs efficiently and informatively. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation. PLDI '92. New York, NY, USA: ACM; 1992. p. 212–23.
[13] Ferrante Jeanne, Ottenstein Karl J, Warren Joe D. The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 1987; 9(July (3)):319–49.

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001

[14] Hardekopf Ben, Lin Calvin. Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL '09. New York, NY, USA: ACM; 2009. p. 226–38.
[15] Hardekopf Ben, Lin Calvin. Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the symposium on code generation and optimization; 2011.
[16] Jhee Yongin, Jin Minsik, Jung Yungbum, Kim Deokhwan, Kong Soonho, Lee Heejong, et al. Abstract interpretation+impure catalysts: our Sparrow experience. In: Presentation at the workshop of the 30 years of abstract interpretation, San Francisco; January 2008. ⟨ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf.
[17] Johnson Richard, Pingali Keshav. Dependence-based program analysis. In: Proceedings of the ACM SIGPLAN 1993 conference on programming language design and implementation, PLDI '93. New York, NY, USA: ACM; 1993. p. 78–89.
[18] Lee Woosuk, Lee Wonchan, Yi Kwangkeun. Sound non-statistical clustering of static analysis alarms. In: VMCAI 2012: 13th international conference on verification, model checking, and abstract interpretation. Lecture notes in computer science. London, UK: Springer; 2012.
[19] Mauborgne Laurent, Rival Xavier. Trace partitioning in abstract interpretation based static analyzers. In: Sagiv M, editor. Proceedings of European symposium on programming. Lecture notes in computer science, vol. 3444. London, UK: Springer-Verlag; 2005. p. 5–20.
[20] Miné A. The octagon abstract domain. Higher-Order Symb Comput 2006;19(1):31–100.
[21] Oh Hakjoo. Large spurious cycle in global static analyses and its algorithmic mitigation. In: Proceedings of the Asian symposium on programming languages and systems. Lecture notes in computer science, vol. 5904. London, UK: Springer-Verlag; December 2009. p. 14–29.
[22] Oh Hakjoo, Brutschy Lucas, Yi Kwangkeun. Access analysis-based tight localization of abstract memories. In: VMCAI 2011: 12th international conference on verification, model checking, and abstract interpretation. Lecture notes in computer science, vol. 6538. London, UK: Springer; 2011. p. 356–70.
[23] Oh Hakjoo, Heo Kihong, Lee Wonchan, Lee Woosuk, Yi Kwangkeun. 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.
[24] Oh Hakjoo, Yi Kwangkeun. Access-based localization with bypassing. In: Proceedings of the Asian symposium on programming languages and systems. Lecture notes in computer science, vol. 7078. London, UK: Springer-Verlag; December 2011. p. 50–65.
[25] Oh Hakjoo, Yi Kwangkeun. Access-based abstract memory localization in static analysis. Science of computer programming; 2013.
[26] Pingali Keshav, Bilardi Gianfranco. Optimal control dependence computation and the roman chariots problem. ACM Trans Program Lang Syst 1997;19(May):462–91.
[27] Ramalingam G. On sparse evaluation representations. Theor Comput Sci 2002;277(1–2):119–47.
[28] Reif John H, Lewis Harry R. Symbolic evaluation and the global value graph. In: Proceedings of the 4th ACM SIGPLAN-SIGACT symposium on principles of programming languages; 1977. p. 104–18.
[29] Sagiv Mooly, Reps Thomas, Horwitz Susan. Precise interprocedural dataflow analysis with applications to constant propagation. Theor Comput Sci 1996;167(October (1–2)):131–70.
[30] Shapiro Marc, Horwitz Susan. The effects of the precision of pointer analysis. In: Proceedings of the international symposium on static analysis; 1997. p. 16–34.
[31] Venet Arnaud, Brat Guillaume. 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. New York, NY, USA: ACM; 2004. p. 231–42.
[32] Wegman Mark N, Zadeck F Kenneth. Constant propagation with conditional branches. ACM Trans Program Lang Syst 1991;13(April):181–210.

Please cite this article as: Ko Y, et al. A sparse evaluation technique for detailed semantic analyses. Computer Languages, Systems & Structures (2014), http://dx.doi.org/10.1016/j.cl.2014.05.001