A posteriori soundness for non-deterministic abstract interpretations*

Matthew Might[1] and Panagiotis Manolios[2]

[1] University of Utah, Salt Lake City, Utah, USA, might@cs.utah.edu
[2] Northeastern University, Boston, Massachusetts, USA, pete@ccs.neu.edu

Abstract. An abstract interpretation's resource-allocation policy (*e.g.*, one heap summary node per allocation site) largely determines both its speed and precision. Historically, context has driven allocation policies, and as a result, these policies are said to determine the "context-sensitivity" of the analysis. This work gives analysis designers newfound freedom to manipulate speed and precision by severing the link between allocation policy and context-sensitivity: abstract allocation policies may be unhinged not only from context, but also from even a predefined correspondence with a concrete allocation policy. We do so by proving that abstract allocation policies can be made non-deterministic without sacrificing correctness; this non-determinism permits precision-guided allocation policies previously assumed to be unsafe. To prove correctness, we introduce the notion of *a posteriori* soundness for an analysis. A proof of *a posteriori* soundness differs from a standard proof of soundness in that the abstraction maps used in an *a posteriori* proof cannot be constructed until *after* an analysis has been run. Delaying construction allows them to be built so as to justify the decisions made by non-determinism. The crux of the *a posteriori* soundness theorem is to demonstrate that a justifying abstraction map can *always* be constructed.

1 Introduction

When engineering a static analysis, better speed and higher precision are principal goals. In abstract interpretation, speed and precision are a function of the abstract allocation policy. By *abstract allocation policy*, we mean the procedure by which an abstract interpretation chooses a resource from a pool of abstract resources during the transition from one abstract state to another. To ground this discussion with some specifics, examples of an abstract resource include abstract environment bindings (for environment analyses [17–19, 21]), abstract heap addresses (for alias and shape analyses [2, 4, 22]), abstract contours (for flow analyses [1, 15, 16, 19, 23–25]), and abstract time-stamps (for frame-string analyses [11, 17, 18, 20]).

The abstract allocation policy determines how the abstract state-space partitions the concrete state-space; likewise, a given partitioning uniquely determines

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

an abstract allocation strategy. (More directly, this policy determines the relationship between the concrete and abstract instances of objects like stores, heaps and environments.) A fast, precise analysis needs an allocation policy which summarizes concrete resources that behave alike to the same abstract resource, but which summarizes concrete resources that behave differently to separate abstract resources.

The original motivation for this work came from frustration with context-sensitive policies: for the best result, one must choose the context-sensitive policy whose heuristic is geared toward the behavior of the program under consideration. We wanted to know whether abstract allocation policies could be made sensitive to precision rather than context—whether it is sound to make allocations based purely on precision, or not. Thus, this work begins by asking a general question: what *are* the fundamental, necessary constraints which all abstract allocation policies must obey. Specifically, we want to know whether an abstract allocation policy must directly simulate the concrete allocation policy in order to prove soundness.

Our pursuit of the constraints on abstract allocation policies ends with an unanticipated result: there are no such constraints. To demonstrate completeness, we prove that even the allocation policy which is fully non-deterministic is safe. Central to this result is the criterion of and a proof technique for "*a posteriori* soundness."

1.1 *A priori* soundness and context-sensitivity

Frequently, abstract resources are selected as a function of the context of the current state, *e.g.*, the current program counter [3], the last k call sites [24], the let-polymorphism of the current function [25], the Cartesian product of argument types [1]. Context has been popular in designing allocation policies because context serves as a reasonably good heuristic for data usage: data structures allocated in the same context (*e.g.*, the same call site, the same stack frame) tend to have similar usage patterns. Context-sensitive policies bleed precision and speed to the extent that they tend to split like resources across several abstract resources, *e.g.*, 1CFA [24], or tend to associate unlike resources as a single abstract resource, *e.g.*, CPA [1].

In an attempt at better performance, one might ask whether allocation policies can be hybridized and proven sound, so that the strengths of the two policies can be combined. For simplistic hybrids, such as the natural "Cartesian product" of two policies, the answer is yes; however, such a hybridization also combines their weaknesses—the splitting tendencies of the two policies multiply: precision goes up, but so does analysis time. If an analysis designer were to compensate for this splitting by making the allocation policy *adaptive*, then proving soundness is suddenly murkier, and the answer *seems* to be no. By *adaptive*, we mean that the allocation policy is allowed to directly consider the ramifications upon the precision or speed of the analysis when selecting an abstract resource to allocate during a transfer function; adaptive behavior is in contrast to the standard behavior of choosing an abstract resource based on context.

The reason that adaptive behavior seems unsafe is that, under the standard correctness regime, abstract allocation policies have to be a simulation of a concrete allocation policy. Thus, if an abstract allocation policy is informed by the precision of the analysis, that information must be available to the concrete allocation policy, so that the two may remain in sync. In general, of course, this is not possible; the concrete execution has perfect precision, and abstractions of it can have myriad degrees of coarseness.

Example Consider a concrete store σ and two abstractions thereof, $\hat{\sigma}$ and $\hat{\sigma}'$:

$$\begin{array}{lll} \sigma(1) = 0 & \hat{\sigma}(\hat{\ell}_1) = \{0, 3\} & \hat{\sigma}'(\hat{\ell}_1) = \{0, 3\} \\ \sigma(2) = 3 & & \\ \sigma(3) = 4 & \hat{\sigma}(\hat{\ell}_2) = \{4\} & \hat{\sigma}'(\hat{\ell}_2) = \{4, 5, 7\} \end{array}$$

Suppose that, in an effort to improve precision, an abstract allocation policy always chose the abstract address with the smallest set of abstract values (as opposed to, for instance, picking the label of the current call site). Under this policy, the next abstract address for allocation would be $\hat{\ell}_2$ if the simulation has $\hat{\sigma}$ as its current store, and $\hat{\ell}_1$ if the simulation has $\hat{\sigma}'$ as its current store. It is *impossible* to define a concrete policy that justifies this behavior, because it cannot know whether to pick a concrete address that abstracts to $\hat{\ell}_1$ or to $\hat{\ell}_2$. □

The inability of the abstract allocation policy to deviate from the concrete allocation policy (or, *vice versa*) is embedded in the established process for proving soundness in an abstract interpretation. It is an artifact of the standard process, which is as old as the Cousots' original framework [6, 7]. We can abbreviate this soundness process as follows:

1. Define a concrete state-space, L.
2. Define a concrete semantics, $f : L \to L$.
3. Define an abstract state-space, $\hat{L}$.
4. Define an abstraction map, $\alpha : L \to \hat{L}$.
5. Define an abstract semantics, $\hat{f} : \hat{L} \to \hat{L}$.
6. Prove abstract semantics $\hat{f}$ simulates concrete semantics f under map α.

For the duration of this work, we term this process the *a priori* soundness process, because the abstraction map α is constructed before the analysis is run.

1.2 *A posteriori* soundness and non-determinism

This work presents a more flexible soundness process—the *a posteriori* soundness process—in which the abstraction map, and hence the soundness of the analysis, is not constructed until *after* the analysis has been computed. We do so in order to circumvent the excessive strictness that is the byproduct of the *a priori* soundness process. With *a posteriori* soundness, one can hybridize allocation policies *and* make them adaptive. Most generally, we will be able to make abstract allocation policies non-deterministic; the immediate consequence—that there are no unsound abstract allocation policies—is an unexpected result.

The *a posteriori* soundness process can be summarized as follows:

1. Define a concrete state-space, L.
2. Define a concrete semantics, $f : L \to L$.
3. Define an abstract state-space, $\hat{L}$.
4. Define a non-deterministic abstract semantics, $\hat{f} : \hat{L} \to \mathcal{P}(\hat{L})$.
5. Execute the abstract semantics to produce an abstract transition graph.
6. Construct an abstraction map, $\alpha : L \to \hat{L}$, such that the abstract transition graph simulates the concrete semantics f under the map α.

Proving *a posteriori* soundness then reduces to proving that no matter how the abstract transition graph evolves from the abstract semantics, it is *always* possible to construct a justifying abstraction map.

1.3 Contributions

This work makes the following contributions:

1. The concept of allocation-policy-factored semantics.
2. A framework for non-deterministic abstract interpretation.
3. The correctness proof technique of *a posteriori* soundness.
4. An instance of the framework for higher-order control flow analysis: ∃CFA.
5. A discussion of allocation policies outside the bounds of context-sensitivity.

2 Policy-factored concrete semantics

The goal in this work is to reason about the limits of allocation policies. The first step, then, is to isolate and factor out allocation policies from semantics. Given a concrete state-space Σ, a semantics can be defined by means of a small-step transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$, or congruently, as a transfer function, $g : \Sigma \rightharpoonup \Sigma$. A *policy-factored* transfer function accepts a state and produces a partial function that takes a concrete "locative" from a set L to the next state: $f : \Sigma \to L \rightharpoonup \Sigma$. The set of locatives L denotes a pool of allocatable objects. We use the term *locative* to generalize over entities such as environment bindings, store locations, contours and time-stamps. Given a concrete semantics g and a factored semantics f, the allocation policy is any function $\pi : \Sigma \to L$ constrained so that:

$$g(\varsigma) = f(\varsigma)(\pi(\varsigma)).$$

Example Consider the one-instruction (Turing-incomplete) language Malloc described by the following grammar:

$$s \in \mathsf{Stmt} ::= \mathit{lab} : \mathit{var} \texttt{ := malloc()}$$

where $\mathit{lab} \in \mathsf{Lab}$ denotes labels on instructions and $\mathit{var} \in \mathsf{Var}$ denotes variables. A state consists of a list of statements, an environment and a store:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Stmt}^* \times \mathit{Env} \times \mathit{Store} \\ \eta \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathbb{N} \\ \sigma \in \mathit{Store} &= \mathbb{N} \rightharpoonup \{0, 1\}. \end{aligned}$$

And the state-to-state transfer function $g : \Sigma \rightharpoonup \Sigma$ just makes allocations:

$$g([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \eta, \sigma) = (\boldsymbol{s}, \eta[var \mapsto n'], \sigma[n' \mapsto 0]),$$

where the address n' is the lowest unused value in the store: $\max(dom(\sigma)) + 1$.

The policy-factored formulation of this semantics is the function f:

$$f([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \eta, \sigma)(\ell) = \begin{cases} (\boldsymbol{s}, \eta[var \mapsto \ell], \sigma[\ell \mapsto 0]) & \ell \notin dom(\sigma) \\ undefined & \ell \in dom(\sigma). \end{cases}$$

Setting $f(\varsigma)(\pi(\varsigma)) = g(\varsigma)$ and solving, we find the set of locatives, $L = \mathbb{N}$, and the allocation policy function $\pi : \Sigma \to L$:

$$\pi(\boldsymbol{s}, \eta, \sigma) = \max(dom(\sigma)) + 1.$$

□

A concrete semantics must also specify an initial state ς_0. The output of an unfactored semantics is a (possibly infinite) sequence of states $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$, such that $\varsigma_{i+1} = g(\varsigma_i)$. The output of a factored semantics is a (possibly infinite) sequence of states $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$ coupled with a sequence of locatives $\boldsymbol{\ell} = \langle \ell_0, \ell_1, \ldots \rangle$, such that $\varsigma_{i+1} = f(\varsigma_i)(\ell_i)$.

3 Policy-factored abstract semantics

Now that we have created a policy-factored concrete semantics, we can create the machinery central to the subject of this work: the policy-factored abstract semantics. Given an abstract state-space $\hat{\Sigma}$, an abstract semantics can be defined through a transition relation $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, or congruently, an abstract transfer function $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$. Generalizing the abstract transfer function, we can create a policy-factored abstract transfer function, $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$. The factored function takes an abstract state to a *set* of functions; each of these functions takes an abstract locative to a subsequent abstract state.

The factored abstract $\hat{f}$ semantics and the abstract allocation policy function $\hat{\pi} : \hat{\Sigma} \to \hat{L}$ are constrained by the equation:

$$\hat{g}(\hat{\varsigma}) = \bigcup_{\hat{h} \in \hat{f}(\hat{\varsigma})} \hat{h}(\hat{\pi}(\hat{\varsigma})).$$

An abstract semantics, factored or otherwise, must also specify an initial abstract state $\hat{\varsigma}_0 \in \hat{\Sigma}$.

Example Using the MALLOC language from before, we can create an abstract semantics. An abstract state is a sequence of statements, an abstract environment, and an abstract store:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Stmt}^* \times \widehat{Env} \times \widehat{Store} \\ \hat{\eta} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \mathsf{Lab} \\ \hat{\sigma} \in \widehat{Store} &= \mathsf{Lab} \to \mathcal{P}\left(\{0, 1\}\right). \end{aligned}$$

A standard abstract-address-per-point, context-sensitive transfer function is the function $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$:

$$\hat{g}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = \{(\boldsymbol{s}, \hat{\eta}[var \mapsto lab], \hat{\sigma} \sqcup [lab \mapsto \{0\}])\}.$$

Policy-factoring this function yields $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$:

$$\hat{f}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = \{\lambda\hat{\ell}.(\boldsymbol{s}, \hat{\eta}[var \mapsto \hat{\ell}], \hat{\sigma} \sqcup [\hat{\ell} \mapsto \{0\}])\}.$$

(Both $\hat{g}$ and $\hat{f}$ return the empty set for empty statement sequences.) Solving for the abstract allocation policy function $\hat{\pi} : \hat{\Sigma} \to \hat{L}$, we get:

$$\hat{\pi}([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \hat{\eta}, \hat{\sigma}) = lab,$$

and for the set of abstract locatives, we have that $\hat{L} = \mathsf{Lab}$. □

4 Non-deterministic abstract interpretation

Factoring out allocation policies makes it possible to describe a framework for abstract interpretation that encompasses all conceivable allocation policies—by making the abstract allocation policy "function" non-deterministic. Of course, the adaptive or precision-sensitive allocation policies we mentioned in the introduction are included under the label *all conceivable*. More specifically, with each application of the transfer function, this framework chooses the abstract locative non-deterministically.

The result of a non-deterministic abstract interpretation is an abstract-locative-labeled transition graph between abstract states:

Definition 1. *An **abstract transition graph** is a labeled graph $(\hat{S}, \rightsquigarrow)$ where:*

- *$\hat{S} \subseteq \hat{\Sigma}$ is a subset of states, and*
- *$(\rightsquigarrow) \subseteq \hat{S} \times \hat{L} \times \hat{S}$ is a set of edges labeled by abstract locatives.*

In lieu of defining an algorithm for computing a non-deterministic abstract interpretation, we define the result of such an abstract interpretation as a *closed* abstract transition graph. A transition graph is closed if it accounts for all abstract transitions from each state:

Definition 2. *An abstract transition graph $(\hat{S}, \rightsquigarrow)$ is **closed** under a policy-factored transfer function $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$ iff for each state $\hat{\varsigma} \in \hat{S}$, for each state-generator $\hat{h} \in \hat{f}(\hat{\varsigma})$, there exists a locative $\hat{\ell}$ such that:*

$$\hat{\varsigma} \overset{\hat{\ell}}{\rightsquigarrow} \hat{h}(\hat{\varsigma})(\hat{\ell}).$$

A *least closed graph*, which contains no closed subgraphs, is preferred (but not required) as the result of a static analysis.

With non-deterministic abstract interpretation defined, our new intermediate task is to define simple, liberal criteria that must hold between concrete and abstract policy-factored transfer functions, so that when this condition holds, any closed abstract transition graph represents a simulation of the concrete execution. The next section reviews the standard criteria for the correctness of context-sensitive policies, in preparation for this generalization.

5 The *a priori* simulation criterion

On the road to constructing criteria for policy-factored abstract transfer functions that guarantee simulation, it is illustrative to review the inductive step of an *a priori* proof of soundness. Proving the soundness of an ordinary abstract interpretation reduces to showing that its abstract transfer function simulates the concrete transfer function with respect to some abstraction map $\alpha : \Sigma \to \hat{\Sigma}$. More formally:

Definition 3 (Transfer function simulation). *The abstract transfer function* $\hat{g} : \hat{\Sigma} \to \mathcal{P}\left(\hat{\Sigma}\right)$ ***simulates*** *the concrete transfer function* $g : \Sigma \rightharpoonup \Sigma$ *with respect to the abstraction map* $\alpha : \Sigma \to \hat{\Sigma}$ *iff*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$$

implies

$$\{\alpha(g(\varsigma))\} \sqsubseteq \hat{g}(\hat{\varsigma}).$$

For context-sensitive analyses, it is standard to have a lemma while proving simulation that shows that the abstract allocation policy function is a simulation of the concrete allocation policy function, or more formally:

Definition 4 (Policy simulation). *The abstract policy function* $\hat{\pi} : \hat{\varsigma} \to \hat{L}$ ***simulates*** *the concrete policy function* $\pi : \Sigma \to L$ *with respect to the abstraction maps* $\alpha : \Sigma \to \hat{\Sigma}$ *and* $\alpha_L : L \to \hat{L}$ *iff*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$$

implies

$$\alpha_L(\pi(\varsigma)) \sqsubseteq \hat{\pi}(\hat{\varsigma}).$$

In some frameworks, such as Shivers's formulation of k-CFA [24], this policy simulation lemma is an explicit requirement.

Example Considering the running Malloc example again reveals much about how analysis designers tinker with the *concrete* semantics to "engineer" the correctness of their *abstract* allocation policies. With the semantics as formulated, we would have to define a locative abstraction map $\alpha_L : \mathbb{N} \to \mathsf{Lab}$ that can

satisfy the policy function simulation requirement. At first glance, this seems awkward—how can we map from a natural number, used as a concrete address, to the label that was used during the abstract interpretation? There doesn't seem to be enough information within the natural number to do so. (Later, we'll show how this can be done with *a posteriori* soundness.) The long-time solution for analysis designers has been to encode the requisite information inside concrete locatives. In this case, the concrete semantics would be modified to use the Cartesian product of labels and naturals for the set of locatives: $L = \mathsf{Lab} \times \mathbb{N}$, yielding:

$$\pi([\![lab : var \texttt{ := malloc()}]\!] : \boldsymbol{s}, \eta, \sigma) = (lab, 1 + \max\{n : (\_, n) \in dom(\sigma)\}).$$

With this reformulation of the concrete semantics, a suitable locative-abstraction map is easily defined:

$$\alpha_L(lab, n) = lab.$$

Now the *a priori* policy-simulation requirement is easy to prove.

A side benefit of proving *a posteriori* soundness is that the concrete semantics do not require reformulation, thereby obviating the need for a proof of equivalence between the original and the re-engineered concrete semantics. □

6 Policy-factored abstraction map

We are closing on our intermediate task of defining a condition on policy-factored semantics that guarantees simulation under non-determinism. Before we can state this condition formally, we need to create a policy-factoring of abstraction maps.

An ordinary proof of soundness for abstract interpretation requires a state-wise abstraction map $\alpha : \Sigma \to \hat{\Sigma}$ to express the relationship between the concrete and abstract domains. In order to allow a non-deterministic abstract interpretation, the proof delays the construction of this map until after the analysis has run. Instead, non-deterministic abstract interpretation employs a policy-factored abstraction map $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$; this function takes an abstraction map over locatives to produce an abstraction map over states.

No further policy-factoring of the semantics is required at this point. Note that the lattice relations and operations on states—($\sqsubseteq$), ($\sqcup$) and ($\sqcap$)—do not require factoring since they operate purely in the abstract state-space.

Example Returning to the MALLOC example once again, the factored abstraction map on states is $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$:

$$\begin{aligned}
\beta(\alpha_L)(\boldsymbol{s}, \eta, \sigma) &= (\boldsymbol{s}, \beta_{Env}(\alpha_L)(\eta), \beta_{Store}(\alpha_L)(\sigma)) \\
\beta_{Env}(\alpha_L)(\eta) &= \lambda var.\alpha_L(\eta(var)) \\
\beta_{Store}(\alpha_L)(\sigma) &= \lambda\hat{\ell}. \bigsqcup_{\alpha_L(\ell) \sqsubseteq \hat{\ell}} \{\sigma(\ell)\}.
\end{aligned}$$

Dropping in the locative-abstraction map from the previous example yields the expected unfactored abstraction map on states: $\alpha = \beta(\alpha_L)$ □

7 The dependent simulation condition

We finally have the machinery required in order to describe a general, liberal condition under which a non-deterministic abstract interpretation is correct: the dependent simulation condition.

Definition 5. *The policy-factored abstract transfer function* $\hat{f} : \hat{\Sigma} \to \mathcal{P}(\hat{L} \to \hat{\Sigma})$ *is a* ***dependent simulation*** *of the policy-factored concrete transfer function* $f : \Sigma \to L \rightharpoonup \Sigma$ *under the factored abstraction map* $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$ *iff, for all locative abstraction maps* $\alpha_L : L \to \hat{L}$*, if*

$$\beta(\alpha_L)(\varsigma) \sqsubseteq \hat{\varsigma},$$

then for any locative ℓ *and any abstract locative* $\hat{\ell}$*, there exists a state-generator* $\hat{h} \in \hat{f}(\hat{\varsigma})$ *such that:*

$$\beta(\alpha_L[\ell \mapsto \hat{\ell}])(f(\varsigma)(\ell)) \sqsubseteq \hat{h}(\hat{\ell}).$$

8 The *a posteriori* soundness theorem

Having defined the dependent simulation condition, it is now possible to prove that a non-deterministic abstract interpretation satisfying this condition is correct, thereby demonstrating that there is no such thing as an illegal abstract allocation policy. A standard proof of soundness is not possible in this case: the abstract allocation policy must simulate the concrete allocation policy, but we cannot describe the abstraction map in advance when the abstract policy is non-deterministic.

First, we must define the concept of a sound simulation for abstract transition graphs over concrete executions.

Definition 6. *An abstract transition graph* $(\hat{S}, \rightsquigarrow)$ *is a* ***sound simulation*** *of a sequence of states* $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$ *under the abstraction map* $\alpha : \Sigma \to \hat{\Sigma}$ *iff*

- *for each* $i \leq \mathit{length}(\boldsymbol{\varsigma})$*:*
$$\{\alpha(\varsigma_i)\} \sqsubseteq \hat{S}, \text{ and}$$
- *for each* $i < \mathit{length}(\boldsymbol{\varsigma})$*:*
$$\{(\alpha(\varsigma_i), \alpha(\varsigma_{i+1}))\} \sqsubseteq (\rightsquigarrow).$$

In other words, each concrete state and each concrete transition is represented in the abstract graph by an abstract state and an abstract edge.

Next, we prove the *a posteriori* soundness theorem. It states that, when the dependent simulation condition is met, a locative-abstraction map that makes a closed abstract transition graph a simulation of a concrete execution must exist.

Theorem 1 (*A posteriori* soundness). *If:*

- $(\boldsymbol{\varsigma}, \boldsymbol{\ell})$ *is a concrete execution for factored transfer function* f*, and*

– *$\hat{f}$ is a dependent simulation of f under factored map β, and*
– *$(\hat{S}, \rightsquigarrow)$ is a closed abstract transition graph for $\hat{f}$ where $\hat{\varsigma}_0 \in \hat{S}$, and*
– *for all maps $\alpha_L : L \to \hat{L}$, $\beta(\alpha_L)(\varsigma_0) \sqsubseteq \hat{\varsigma}_0$,*

then there exists a map $\alpha_L : L \to \hat{L}$ such that the graph $(S, \rightsquigarrow)$ is a sound simulation of the sequence $\boldsymbol{\varsigma}$ under the abstraction map $\beta(\alpha_L)$.

Proof. The proof proceeds by construction of the locative abstraction map. We do so by defining a sequence of abstract states $\hat{\boldsymbol{\varsigma}} = \langle \hat{\varsigma}_0, \hat{\varsigma}_1, \ldots \rangle$, a sequence of abstract locatives $\hat{\boldsymbol{\ell}} = \langle \hat{\ell}_0, \hat{\ell}_1, \ldots \rangle$ and a sequence of partial locative abstraction maps $\boldsymbol{\alpha} = \langle \alpha_0, \alpha_1, \ldots \rangle$ through recurrence equations. We show by induction that $\beta(\alpha_i)(\varsigma_i) \sqsubseteq \hat{\varsigma}_i$.

Let N be the length of the concrete execution sequence $\boldsymbol{\varsigma} = \langle \varsigma_0, \varsigma_1, \ldots \rangle$. For later use, fix a choice function *choose* : $\mathcal{P}\left(\hat{L} \times \hat{\Sigma}\right) \to (\hat{L} \times \hat{\Sigma})$. The initial abstraction map α_0 is defined to be $\bot_{\hat{L}}$ at every point: $\alpha_0 = \lambda \ell . \bot_{\hat{L}}$.

We construct the abstract locative $\hat{\ell}_i$ and the abstract state $\hat{\varsigma}_{i+1}$ simultaneously. Let the set of candidate transitions $C_i \subseteq \hat{L} \times \hat{\Sigma}$ be:

$$C_i = \{(\hat{\ell}, \hat{\varsigma}) : \hat{\varsigma}_i \overset{\hat{\ell}}{\rightsquigarrow} \hat{\varsigma} \text{ and } \beta(\alpha_i[\ell_i \mapsto \hat{\ell}])(\varsigma_i) \sqsubseteq \hat{\varsigma}\}.$$

The set C_i must be non-empty because the graph is closed and the dependent simulation criterion is satisfied. So, we set $(\hat{\ell}_i, \hat{\varsigma}_{i+1}) = \mathit{choose}(C_i)$ and $\alpha_{i+1} = \alpha_i[\ell_i \mapsto \hat{\ell}_i]$. The satisfying locative abstraction map is then:

$$\alpha_L = \lim_{i \to N} \alpha_i.$$

Example We will now construct *a posteriori* locative-abstraction maps for the following Malloc program:

```
L1: x := malloc()
L2: y := malloc()
L3: z := malloc()
```

Using natural numbers for concrete addresses yields the following final state:

$$\varsigma_f = (\langle\rangle, [[\![\mathtt{x}]\!] \mapsto 1, [\![\mathtt{y}]\!] \mapsto 2, [\![\mathtt{z}]\!] \mapsto 3], [1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0]).$$

If a non-deterministic abstract interpretation allocated the abstract locative $\hat{\ell}_1$, then $\hat{\ell}_2$, then $\hat{\ell}_2$, then the locative-abstraction map $\alpha_L : L \to \hat{L}$ would be:

$$\alpha_L(1) = \hat{\ell}_1 \qquad \alpha_L(2) = \hat{\ell}_2 \qquad \alpha_L(3) = \hat{\ell}_2.$$

If, instead, it had allocated $\hat{\ell}_1$, then $\hat{\ell}_2$, then $\hat{\ell}_1$, then the locative-abstraction map would be:

$$\alpha_L(1) = \hat{\ell}_1 \qquad \alpha_L(2) = \hat{\ell}_2 \qquad \alpha_L(3) = \hat{\ell}_1.$$

In each case, the locative-abstraction map leads to simulation. □

9 Example: ∃CFA

To demonstrate the applicability of the *a posteriori* soundness theorem, we construct the generalized, non-deterministic higher-order control-flow analysis (∃CFA) by policy-factoring k-CFA [23, 24]. This leads to a factored concrete transfer function, $f : \Sigma \to L \rightharpoonup \Sigma$, a factored abstract transfer function, $\hat{f} : \hat{\Sigma} \to \mathcal{P}\left(\hat{L} \to \hat{\Sigma}\right)$ and a factored abstraction map, $\beta : (L \to \hat{L}) \to (\Sigma \to \hat{\Sigma})$. An interesting side effect of constructing ∃CFA is that it doubles as a proof of correctness for all existing CFAs (*e.g.*, 0CFA, k-CFA, poly/CFA, CPA) and all future CFAs.

For simplicity, we operate over continuation-passing style (CPS), as described in the following grammar:

$$\begin{aligned} v \in \mathsf{Var} &\quad \text{is a set of identifiers} \\ \lambda \in \mathsf{Lam} &::= (\lambda\ (v_1 \cdots v_n)\ \mathit{call}) \\ f, e \in \mathsf{Exp} &= \mathsf{Var} + \mathsf{Lam} \\ \mathit{call} \in \mathsf{Call} &::= (f\ e_1 \cdots e_n). \end{aligned}$$

A concrete state consists of a call site, a binding environment over variables and a value environment over bindings:

$$\begin{aligned} \varsigma \in \Sigma_{\mathrm{CPS}} &= \mathsf{Call} \times \mathit{BEnv} \times \mathit{VEnv} \\ \\ \rho \in \mathit{BEnv} &= \mathsf{Var} \rightharpoonup L \\ b \in \mathit{Bind} &= \mathsf{Var} \times L \\ \\ \mathit{ve} \in \mathit{VEnv} &= \mathit{Bind} \rightharpoonup D \\ d \in D &= \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{BEnv}. \end{aligned}$$

The policy-factored concrete transfer function f is:

$$f([\![(f\ e_1 \cdots e_n)]\!], \rho, \mathit{ve})(\ell) = (\mathit{call}, \rho'', \mathit{ve}'),$$

defined only if no variable v exists so that $(v, \ell) \in \mathit{dom}(\mathit{ve})$ and where:

$$\begin{aligned} ([\![(\lambda\ (v_1 \cdots v_n)\ \mathit{call})]\!], \rho') &= \mathcal{A}(f, \rho, \mathit{ve}) \\ d_i &= \mathcal{A}(e_i, \rho, \mathit{ve}) \\ \rho'' &= \rho'[v_i \mapsto \ell] \\ \mathit{ve}' &= \mathit{ve}[(v_i, \ell) \mapsto d_i], \end{aligned}$$

where the argument evaluator is $\mathcal{A} : \mathsf{Exp} \to \mathit{BEnv} \rightharpoonup D$:

$$\begin{aligned} \mathcal{A}(\lambda, \rho, \mathit{ve}) &= (\lambda, \rho) \\ \mathcal{A}(v, \rho, \mathit{ve}) &= \mathit{ve}(v, \rho(v)). \end{aligned}$$

The abstract state-space is:

$$\begin{array}{rl}\hat{\varsigma} \in \hat{\Sigma}_{\text{CPS}} & = \mathsf{Call} \times \widehat{BEnv} \times \widehat{VEnv} \\ \\ \hat{\rho} \in \widehat{BEnv} & = \mathsf{Var} \rightharpoonup \hat{L} \\ \hat{b} \in \widehat{Bind} & = \mathsf{Var} \times \hat{L} \\ \\ \widehat{ve} \in \widehat{VEnv} & = \widehat{Bind} \to \hat{D} \\ \hat{d} \in \hat{D} & = \mathcal{P}\left(\widehat{Clo}\right) \\ \widehat{clo} \in \widehat{Clo} & = \mathsf{Lam} \times \widehat{BEnv}.\end{array}$$

According to this, the abstract locatives correspond to the abstract contours of higher-order control-flow analysis.

The policy-factored abstract transfer function $\hat{f}$ is:

$$\hat{f}([\![(f\ e_1 \cdots e_n)]\!], \hat{\rho}, \widehat{ve}) = \{\lambda\hat{\ell}.\mathcal{F}(\widehat{clo})(\hat{\ell}) : \widehat{clo} \in \hat{\mathcal{A}}(f, \hat{\rho}, \widehat{ve})\},$$

where the state finalizer $\mathcal{F} : \widehat{Clo} \to \widehat{L} \to \hat{\Sigma}$ is:

$$\begin{array}{rl}\mathcal{F}([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \hat{\rho}')(\hat{\ell}) = (call, \hat{\rho}'', \widehat{ve}'), \text{ where:} \\ \hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\rho}, \widehat{ve}) \\ \hat{\rho}'' = \hat{\rho}'[v_i \mapsto \hat{\ell}] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \hat{\ell}) \mapsto \hat{d}_i],\end{array}$$

where the abstract argument evaluator is $\hat{\mathcal{A}} : \mathsf{Exp} \to \widehat{BEnv} \to \hat{D}$:

$$\begin{array}{l}\hat{\mathcal{A}}(\lambda, \hat{\rho}, \widehat{ve}) = \{(\lambda, \hat{\rho})\} \\ \hat{\mathcal{A}}(v, \hat{\rho}, \widehat{ve}) = \widehat{ve}(v, \hat{\rho}(v)).\end{array}$$

The appropriate policy-factored abstraction map $\beta : (L \to \hat{L}) \to \Sigma \to \hat{\Sigma}$ walks component-wise over the state-space:

$$\begin{array}{rl}\beta(\alpha_L)(call, \rho, ve) & = (call, \beta_{BEnv}(\alpha_L)(\rho), \beta_{VEnv}(\alpha_L)(ve)) \\ \beta_{BEnv}(\alpha_L)(\rho)(v) & = \alpha_L(\rho(v)) \\ \beta_{VEnv}(\alpha_L)(ve))(v, \hat{\ell}) & = \bigsqcup_{\alpha_L(\ell)=\hat{\ell}} \{\beta_{Clo}(\alpha_L)(ve(v, \ell))\} \\ \beta_{Clo}(\alpha_L)(\lambda, \rho) & = (\lambda, \beta_{BEnv}(\alpha_L)(\rho)).\end{array}$$

10 Adaptive allocation policies

One of the payoffs for proving all conceivable allocation policies correct is in the ability to make allocation policies *adaptive*, or more precisely, precision-sensitive.

A precision-sensitive abstract allocation policy makes allocation decisions based on the perceived effect upon the precision of the analysis.

One way to make a context-sensitive allocation policy into a greedy precision-sensitive policy is to augment it with a reserve pool of m abstract locatives, where m is a fixed cap. Abstract locatives in the reserve pool are not associated with a particular context; they are allocated as necessary to prevent excessive merging. For example, if the default abstract locative to be allocated would cause a closure over λ_{42} to coexist in the same abstract store slot as a closure over λ_{314}, then an adaptive analysis can allocate from the reserve pool of abstract locatives to prevent the merge, so long as the reserve pool is not yet exhausted. Adding a reserve pool of abstract locatives to an existing context-sensitive analysis is a simple way to alleviate the damage to precision from places where the context-sensitive heuristic causes excess merging. Enlarging the reserve pool is an effective way of gradually improving the precision of the analysis.

Adaptive analysis alleviates excess splitting by looking for abstract locatives which, upon allocation, cause the least change to the abstract store. For example, if the default abstract locative would give a fresh slot to a closure over λ_{42}, when another slot already contains a closure over λ_{42}, the adaptive analysis may opt to allocate the locative already in use.

Adaptive allocation policies, which are not provably correct under *a priori* soundness, highlight the practical advantages of non-determinism in abstract interpretation.

11 Related work

This work taps into the foundations of the Cousots' work on abstract interpretation [6, 7]. The standard soundness recipe we presented is a simplification of the soundness regime presented throughout their work [5, 8]. The use of *a posteriori* abstraction maps is a simple way of extending their framework to allow a practical degree of non-determinism in abstract interpretation.

This work should not be confused with the body of work on the (deterministic) abstract interpretation of non-deterministic systems [9, 13]. However, it is likely that non-deterministic abstract interpretation of non-deterministic systems will lead to considerable gains in precision. This work should also not be confused with random interpretation [10], which is unsound. Our work is related in that we enable probabilistic abstract interpretation, but our work retains soundness.

This work impacts the large body of work on alias and shape analysis [2–4, 12, 22] by liberating these analyses from the needless rigidity imposed by *a priori* abstraction maps.

This work also directly impacts higher-order relatives of alias and shape analysis, environment analysis [14, 17–19, 21] and control-flow analysis [15, 16, 23, 24], by expanding the set of contour-allocation schemes.

12 Conclusions and Future Work

We have presented a framework for enabling sound non-deterministic abstract interpretations. We introduced non-determinism into allocation policies in order to free analyses from the rigidity of *a priori* abstraction maps. By proving the correctness of the non-deterministic framework using the novel proof technique of *a posteriori* abstraction maps, we have proven that all conceivable abstract allocation policies are correct. We discussed a practical benefit: that allocation policies may be made adaptive with respect to analytic precision, a behavior which cannot be proven sound under the Cousots' standard correctness framework. And, we instantiated this framework to create a non-deterministic flow analysis: ∃CFA.

For future work, we plan to explore precision-sensitive allocation where abstract locatives are allocated probabilistically, according to evolving distributions that tend toward "do not allocate" in the limit. We also plan to investigate the issue of optimality. For example, for an alias analysis, a good metric would be the average size of an abstract value set in the abstract store; the equivalent metric for a CFA would be the average size of a flow set. For a fixed set of n abstract locatives and a given program, there must exist optimal allocation policies which minimize this metric. With a notion of optimality, we can begin to ask whether there are fundamental bounds on precision, and whether an optimal allocation policy can be computed without resorting to exhaustive search.

References

1. Agesen, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP 1995* (1995), pp. 2–26.
2. Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
3. Balakrishnan, G., and Reps, T. Recency-abstraction for heap-allocated storage. In *Proceedings of the Static Analysis Symposium* (Seoul, Korea, 2006).
4. Chase, D. R., Wegman, M., and Zadeck, F. K. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, New York, June 1990), pp. 296–310.
5. Cousot, P. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science 6* (1997). URL: `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages.
6. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.
7. Cousot, P., and 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, Texas, 1979), ACM Press, New York, NY, pp. 269–282.
8. Cousot, P., and Cousot, R. Abstract interpretation frameworks. *Journal of Logic and Computation 2*, 4 (Aug. 1992), 511–547.

9. Gallagher, J., Gallagher, J. P., Puebla, G., and Puebla, G. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *In Fourth International Symposium on Practical Aspects of Declarative Languages, number 2257 in LNCS* (2002), Springer-Verlag, pp. 243–261.
10. Gulwani, S. Program analysis using random interpretation. In *Ph.D. Dissertation, UC-Berkeley* (2005).
11. Harrison, W. L. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation 2*, 3/4 (Oct. 1989), 179–396.
12. Hudak, P. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, Aug. 1986), pp. 351–363.
13. Huth, M. An abstraction framework for mixed non-deterministic and probabilistic systems. In *Validation of stochastic systems*, vol. 2925. 2004, pp. 419–444.
14. Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. K. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, January 1998), pp. 329–341.
15. Jones, N. D. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming* (London, UK, 1981), Springer-Verlag, pp. 114–128.
16. Jones, N. D., and Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1982), ACM, pp. 66–74.
17. Might, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.
18. Might, M., and Shivers, O. Environment analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006)* (Charleston, South Carolina, January 2006), pp. 127–140.
19. Might, M., and Shivers, O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)* (Portland, Oregon, September 2006), pp. 13–25.
20. Might, M., and Shivers, O. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science 375*, 1–3 (May 2007), 137–168.
21. Might, M., and Shivers, O. Exploiting reachability and cardinality in abstract interpretation. *Journal of Functional Programming* (2008).
22. Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages* (1999), pp. 105–118.
23. Shivers, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, June 1988), pp. 164–174.
24. Shivers, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
25. Wright, A. K., and Jagannathan, S. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*, 1 (January 1998), 166–207.