Tunable Control-Flow Sensitivity for Abstract Interpretation

Ben Hardekopf[†] Berkeley Churchill[†] Vineeth Kashyap[†] Ben Wiedermann[‡]

[†]University of California, Santa Barbara [‡]Harvey Mudd College

benh@cs.ucsb.edu bchurchill@umail.ucsb.edu vineeth@cs.ucsb.edu benw@cs.hmc.edu

Abstract

We design an abstract interpretation framework in which *control-flow sensitivity* (e.g., flow-sensitivity, context-sensitivity, object-sensitivity, property simulation, etc.) is formally expressed as a widening operator on a powerset lattice of abstract states. This widening operator enables tractable abstract interpreters with tunable precision that are based directly on the language's concrete semantics. The analysis designer defines control-flow sensitivity for a particular analysis as an abstraction of the program's execution trace; different abstractions lead to different sensitivities. These trace abstractions can be defined separately from the rest of the analysis, allowing the designer to tune the control-flow precision of the abstract interpreter independently of the rest of the abstract semantics.

We define a general, semantics-agnostic framework for creating these tunable abstract interpreters. We then give a concrete example of how to apply the framework to a specific language and desired analysis—a combination of constant propagation and control-flow analysis for a language with higher-order functions and mutable state. We use our framework to derive six different control-flow sensitivities for our example analysis, all without changing the analysis' abstract semantics. Finally, to demonstrate that our method works on real-world languages, we implement a tunable abstract interpreter for JavaScript and show that we can derive a variety of control-flow sensitivities that yield tractable analyses.

1. Introduction

Program analysis design balances *precision* and *performance*—an analysis must be precise enough for its solution to be useful, yet have sufficient performance as to be practical. An important dimension of this tradeoff is *control-flow sensitivity*: how precisely the analysis adheres to realizable program execution paths. There are a number of existing approaches along this dimension, such as flow-insensitive analysis [2], flow-sensitive maximal-fixed-point [13], callstring context-sensitivity (e.g., *k*-CFA [26]), object sensitivity [19], and more. However, these existing approaches are generally specified semi-formally at best and leave open some important questions:

- What are the semantic foundations of these techniques? In other words, what connection does a particular technique have to the concrete semantics of the program being analyzed?
- What is the space of possible control-flow sensitivities? What ties together all the existing ad-hoc techniques, and what other useful techniques might exist?
- Among these choices, how do we figure out which choice is best for a particular analysis and domain of programs? Trial-and-error (or just plain guessing) is the common approach, and currently the only way to explore different choices is to implement each one from scratch. Is there a better way?

This paper aims to answer these questions. We develop an abstract-interpretation framework in which we derive a space of control-flow sensitivities. This space includes the existing set of control-flow sensitivities mentioned above. We show, using a detailed example, that our method can be used to directly implement a tractable abstract interpreter with tunable control-flow sensitivity (i.e., the sensitivity can be varied without changing either the abstract semantics or the analyzer implementation). Finally, we demonstrate that our method scales to realistic languages by using it to implement a tractable, tunable abstract interpreter for JavaScript and applying that interpreter to a set of benchmarks using a variety of control-flow sensitivities.

1.1 Program Analysis

Program analysis determines invariants about program behavior—things that will always happen during every execution, or that will never happen during any execution. This information is useful for a range of applications, including verification, error-checking, security, optimization, and software engineering. However, determining the exact set of program invariants is undecidable; the analysis would need to explore a potentially infinite domain of possible inputs and an infinite number of possible execution paths. Thus, a sound, tractable analysis *under-approximates* invariants by *over-approximating* program behaviors. Anything that the analysis reports as a program invariant truly does hold across all possible program executions, but some real program invariants may not be found because they are disguised by spurious behaviors.

Under these constraints an analysis designer must juggle three different, and often opposing, analysis characteristics: *soundness* (any reported invariant is actually an invariant); *precision* (the analysis computes a close approximation to the set of real program invariants); and *tractability* (the analysis computes its result using a reasonable amount of resources). All three are desirable, but achieving all three at once is difficult. Designing analyses means making tradeoffs among these three competing goals.

The key to a provably sound analysis is to formally connect the program analysis itself with the concrete behavior of the language being analyzed. This is a defining characteristic of *abstract interpretation* [4, 5, 11]: the analysis designer establishes a mathematical relation between the concrete semantics of the analyzed language and the abstract semantics that defines the program analysis. These abstract semantics must over-approximate both data (the concrete values the program operates on) and control (the execution paths that the program takes) such that the intractable concrete domains are partitioned into tractable abstract domains. Abstract interpretation has a rich set of abstract data domains such as intervals [4], octagons [20], and convex polyhedra [6], but (with a few exceptions discussed in Section 6) there has been little work in the area of control abstractions. Thus, so far, abstract interpretation has had little to say about control-flow sensitivity.

An alternative approach to program analysis, *dataflow analysis* [12, 13, 15], takes a more *laissez-faire* attitude to soundness; there is no formal connection required between the program analysis itself and the concrete semantics of the language being analyzed. This means that there is no formal guarantee that the analysis is sound, but it also means that analysis designers are free to use abstractions that just "seem right" without having to prove that they are correct. This freedom has led to a wide variety of control-

flow sensitivities in the dataflow analysis canon [2, 13, 19, 25, 26]. However, the ad-hoc nature of these approaches means that there is no well-defined space of control-flow abstractions, no understanding of how these various abstractions relate to one another, nor any way to easily tune an analysis among different points in this space.

The goal of our work is to bridge this gap between abstract interpretation and dataflow analysis with respect to control-flow sensitivity—we bring the many useful and well-explored control-flow abstractions from dataflow analysis into the fold of abstract interpretation by giving them a formal semantic foundation. This foundation allows us to go beyond existing abstractions by providing a coherent framework that enables these abstractions to be easily and modularly tuned within a well-defined space.

1.2 Key Insights

The following insights allow us to achieve our goals:

Controlling state-space explosion. Naïvely exploring all possible program behaviors results in an intractably large state-space (either infinite or exponentially large with respect to program size). We can control the size of the state-space by defining a widening operator (in the formal abstract interpretation sense) parameterized by an equivalence relation among states.

Control-flow sensitivity. Given this widening operator, we can specify control-flow sensitivity as a particular equivalence relation defined by abstracting the execution history of a program.

Tunability. The execution history can be abstracted in many different ways. By carefully defining the widening operator we can arbitrarily tune the control-flow sensitivity simply by defining different abstractions of the execution history.

These insights lead not only to a theoretical framework, but also to a usable abstract interpreter implementation, formally based on the concrete semantics, that can directly express a large family of tractable program analyses.

1.3 Contributions

The specific contributions of this paper are:

- A formal, semantics-based description of a large space of control-flow sensitivities, including many sensitivities from the dataflow analysis literature that have been described only semi-formally before now (Section 3).
- An in-depth example (using a language with mutable state and higher-order functions) that uses our method to create a tractable abstract interpreter with tunable control-flow sensitivity (Section 4).
- An evaluation of our method using a tunable, provably sound abstract interpreter for JavaScript on a set of benchmarks with a variety of control-flow sensitivities (Section 5).

These contributions provide a principled approach to the design and implementation of control-flow sensitive analyses. Our JavaScript implementation empirically demonstrates that we can leverage some of the best attributes of abstract interpretation and dataflow analysis to scale powerful and provably sound analyses to one of the most difficult-to-analyze modern languages.

2. Abstract Interpretation Background

In this section we describe how abstract interpretation is used to design a provably-sound program analysis; this presentation is based upon previous descriptions of abstract interpretation [4]. Readers already familiar with abstract interpretation may wish to read this section regardless, as it introduces notation that we will use throughout the paper.

We leave the syntax and semantics of the language being analyzed unspecified except that we assume the semantics is in the form of a state transition system. This assumption is not a requirement of abstract interpretation *per se*, but it is a sensible choice because program analysis is usually concerned with the intensional behavior of the program (i.e., its internal states as it executes).

2.1 Concrete Semantics

We start with a concrete semantics of the language being analyzed. This semantics is in the form of a state transition system:

$$\varsigma \in \Sigma \qquad \text{concrete states}$$
$$\mathcal{F} \in \Sigma \to \Sigma \qquad \text{transition function}$$

We leave the exact nature of a state unspecified; a specific example might be a tuple $\langle \textit{program point, store} \rangle$. Given a program P we wish to discover invariants over all possible executions of P. Thus, abstract interpretation begins by using the concrete semantics to define a (uncomputable) concrete *collecting semantics* $[\![P]\!]$ that describes all possible program behaviors. Typically this is defined as the set of all reachable concrete states starting from a set of initial states Σ_I. We use a least fixed-point operator to formalize this notion. Define the functor $\mathring{\cdot}$ so that for set S and function $\mathcal{F}$:

$$\mathring{\mathcal{F}}(S) = S \cup \mathcal{F}(S)$$

The result of this functor computes the image of the given function and is guaranteed to be extensive: $S \subseteq \mathring{\mathcal{F}}(S)$. Then:

$$[\![P]\!] = \mathbf{lfp}_{\Sigma_I} \mathring{\mathcal{F}}$$

This is known as a *first-order* collecting semantics, because it only collects reachability information and discards any information about the relations between the states (i.e., the transitions between them). The space of possible solutions to this equation forms a lattice $\mathcal{L} = (\mathcal{P}(\Sigma), \subseteq, \cap, \cup)$, where $\subseteq$ is the ordering relation, $\cap$ is the meet operator, and $\cup$ is the join operator.

2.2 Abstract Semantics

The concrete collecting semantics would allow us to derive an exact set of invariants on program behavior, however in general (e.g., for any Turing-complete language) it is not computable. To design a program analysis we create an abstract semantics that over-approximates the behaviors of the concrete semantics and that *is* computable. The abstract semantics is also in the form of a state transition system, however we use a transition *relation* rather than a *function*. This change is necessary because the abstract semantics loses information with respect to the concrete semantics, leading to potential nondeterminism in the abstract state transitions (e.g., whether to take the *true* or *false* branch of a conditional):

$$\hat{\varsigma} \in \Sigma^\sharp \qquad \text{abstract states}$$
$$\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp \qquad \text{transition relation}$$

It is convenient to implicitly lift $\mathcal{F}^\sharp$ to be a function with signature $\mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$ that computes the image of the nondeterministic relation; henceforth we use $\mathcal{F}^\sharp$ to mean this lifted version. The abstract states should form a lattice $\mathcal{L}^\sharp = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$. The first-order abstract collecting semantics is defined as:

$$[\![P]\!]^\sharp = \mathbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp$$

i.e., the set of all reachable abstract states. The powerset of abstract states forms a preorder $(\mathcal{P}(\Sigma^\sharp), \preceq)$ such that for sets $A, B \in \mathcal{P}(\Sigma^\sharp)$:

$$A \preceq B \iff \forall a \in A, \exists b \in B \,.\, a \sqsubseteq b$$

Intuitively $A \preceq B$ means that B contains at least as much information as A, thus A is more precise than B.

An abstract semantics is *sound* if any invariant of the abstract semantics is also an invariant of the concrete semantics. To prove soundness we must establish a formal relation between the concrete and abstract collecting semantics. The basis of this relation is a concretization function $\gamma \in \mathcal{L}^\sharp \to \mathcal{L}$ that maps abstract states to sets of concrete states, meaning that if $\varsigma \in \gamma(\hat{\varsigma})$ then $\hat{\varsigma}$ over-approximates ς. Then $\mathcal{F}^\sharp$ is sound if:

$$\mathcal{F} \circ \gamma \sqsubseteq \gamma \circ \mathcal{F}^\sharp$$

In other words, taking an abstract step over-approximates the result of taking a concrete step. Once this foundation is in place, the soundness of the abstract semantics follows from a fixed-point transfer theorem:

Theorem 1 (FIXED-POINT TRANSFER)**.**

$$\text{If } \Sigma_I \sqsubseteq \gamma(\Sigma_I^\sharp) \text{ then } \mathbf{lfp}_{\Sigma_I} \mathring{\mathcal{F}} \sqsubseteq \gamma\left(\mathbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp\right).$$

If we start with an over-approximation of the concrete initial states, then we end up with an over-approximation of the concrete reachable states—thus any invariants we infer about program behavior under the abstract semantics must be invariants about behavior under the concrete semantics. However, since the set of reachable abstract states over-approximates the set of reachable concrete states, they contain spurious program behaviors that may disguise some real program invariants.

2.3 Abstract Domains and Widening

To ensure that the abstract semantics is computable, the simplest method is to require that the abstract domains used by the abstract states are finite. If the abstract domains are finite then the abstract state-space is finite, and hence the set of reachable states is computable. For example, if concrete states contain integers we could abstract the infinite set of integers $\mathbb{Z}$ with a finite abstract domain such as $\{\top, +, -, 0, \bot\}$—the positive/negative/zero lattice. However, finite abstract domains are not strictly necessary for computability; for example, we could employ a *widening operator* [3, 4].

Suppose that we have an abstract domain $\mathcal{D}$ and function $\mathcal{F} \in \mathcal{D} \to \mathcal{D}$ for which we wish to compute a fixed-point, but $\mathcal{D}$ is infinite or just very large—thus the fixed-point computation either diverges or simply takes too long. We can use a widening operator to either force convergence (for an infinite domain) or accelerate convergence (for a very large domain). The idea is to leap-frog the fixed-point computation up the poset of possible solutions, skipping over potentially infinitely many intermediate points that the normal fixed-point computation would have to visit.

Definition 1 (WIDENING OPERATOR)**.** A function $\bigtriangledown \in \mathcal{P}(\mathcal{D}) \to \mathcal{D}$ over a poset $(\mathcal{D}, \sqsubseteq)$ is a (set) widening operator if:

i. $\forall S \in \mathcal{P}(\mathcal{D})$ if $s \in S$, then $s \sqsubseteq \bigtriangledown(S)$
ii. For all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$, the following increasing chain is not strictly increasing:

$$y_0 = x_0, \ldots, y_i = \bigtriangledown\{\, x_j \mid 0 \le j \le i \,\}, \ldots .$$

Then $\bigtriangledown(S)$ over-approximates every $s \in S$, and if we apply the widening operator at each step of the fixed-point computation we are guaranteed to converge to a solution. Because we are using the widening operator this solution will be an over-approximation of the least fixed-point of the original function without widening. If S contains exactly 2 elements a and b we also use the infix notation $a \bigtriangledown b$ to stand for $\bigtriangledown(\{a, b\})$. For our framework, we require that all domains are equipped with a widening operator $\bigtriangledown$ (if the domain is a lattice with finite ascending chains then the standard lattice join is itself a widening operator).

3. Theoretical Framework

Our framework addresses a problem that is not addressed in the general abstract interpretation literature. Even if the abstract collecting semantics $\mathbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp$ is computable and the abstract data domains (e.g., of integers, strings, functions, etc.) are tractable, the analysis in general remains intractable. The problem is control-flow—specifically the nondeterministic choices that must be made because of the analysis' over-approximations: which branch of a conditional should be taken, whether a loop should be entered or exited, which function should be called, etc. The number of abstract states in the fixed-point grows exponentially with the number of nondeterministic choices.

In this section we explain our method for creating a tractable abstract semantics with tunable control-flow sensitivity. The framework is general and applies to a variety of languages and semantics, thus the description is high-level—for concreteness we give a detailed example of how to use the framework for a specific language, semantics, and program analysis in Section 4. The explanation is in three main parts:

- We discuss some requirements on the form of language semantics necessary to successfully abstract control-flow to attain control-flow sensitivity.
- We describe a widening operator on abstract states that provides a generic "knob" with which we can tune the precision/performance trade-off of the analysis.
- We show that the various control-flow sensitivities described in the dataflow analysis canon are specific instantiations of this knob based on abstracting the trace-based second-order abstract collecting semantics.

3.1 Abstracting Control

We assume that the concrete and abstract states contain some explicit representation of the rest of the computation—i.e., a continuation. This representation can be in the form of a syntactic continuation (e.g., if a program is in continuation-passing style then the "rest of the computation" is given as a closure in the store) or a semantic continuation (e.g., the continuation stack of an abstract machine). Since the abstract states form a lattice, two different states must have a join: some continuation that over-approximates the input continuations. Thus, by joining states we are approximating control as well as data. Dataflow analysis makes a similar assumption when it expects the control-flow graph (CFG) as input to the analysis. The CFG's edges are a kind of continuation and allow the analysis to over-approximate program paths.

Some forms of semantics do not meet this requirement. For example, big-step and small-step structural operational semantics implicitly embed the continuations in the semantic rules. Direct-style denotational semantics similarly embeds this information in the translation to the underlying meta-language. Because there is no explicit handle on the control-state, there is no way to abstract and approximate the control-state: the analysis is stuck with whatever control the original semantics specifies. Some limited forms of control-flow sensitivity are still possible (e.g., flow-sensitive maximal fixed-point), but many others are not (e.g., k-CFA).

3.2 Widening Abstract States

We can limit the exponential growth of abstract states during the fixed-point computation by sacrificing precision and joining abstract states together, over-approximating the original fixed-point. However, we need a way to control *which* states to join. We proceed by (1) creating abstract configurations that pair abstract states with an unspecified but finite *knob* domain $\mathcal{K}$; and (2) describing a widening operator $\bigtriangledown^\sharp$ that uses $\mathcal{K}$ to control the joining of abstract

states. This approach results in a tunable analysis parameterized by the definition of $\mathcal{K}$ and its transition function δ.

3.2.1 Abstract Configurations

Let $\mathcal{K}$ be an unspecified but finite set; this set does not need to be ordered. We define abstract configurations C and their transition function $\mathcal{G} \in \mathcal{P}(C) \to \mathcal{P}(C)$ as:

$$\begin{aligned} \langle \hat{\varsigma}, \kappa \rangle \in C &= \Sigma^\sharp \times \mathcal{K} && \text{configurations} \\ \mathcal{G}(\langle \hat{\varsigma}, \kappa \rangle) &= \left\{ \langle \hat{\varsigma}', \kappa' \rangle \;\middle|\; \hat{\varsigma}' \in \mathcal{F}^\sharp(\hat{\varsigma}) \right\} && \text{transition function} \\ C_I &= \left\{ \langle \hat{\varsigma}, \kappa_I \rangle \;\middle|\; \hat{\varsigma} \in \Sigma^\sharp_I \right\} && \text{initial configurations} \end{aligned}$$

where $\mathcal{G}$ is implicitly lifted to operate on sets. The κ' components in $\mathcal{G}$'s result are constrained by an unspecified knob transition function δ (δ's precise definition is irrelevant to the results in this section). The initial configurations use a distinguished initial element $\kappa_I \in \mathcal{K}$. If we compute the set of reachable abstract configurations using $\mathcal{G}$ and then project out only the $\Sigma^\sharp$ component of the resulting tuples[1], we get exactly the set of reachable abstract states using $\mathcal{F}^\sharp$:

Lemma 1. $\mathbf{lfp}^{\circ}_{\Sigma^\sharp_I} \mathcal{F}^\sharp = \left\{ \pi_{\hat{\varsigma}}(c) \;\middle|\; c \in \mathbf{lfp}^{\circ}_{C_I} \mathcal{G} \right\}$.

Proof. By definition of $\mathcal{G}$. □

3.2.2 The Widening Operator

We define an equivalence relation $\sim$ on abstract configurations so that two configurations are equivalent iff they have the same knob:

$$\langle \hat{\varsigma}_1, \kappa_1 \rangle \sim \langle \hat{\varsigma}_2, \kappa_2 \rangle \iff \kappa_1 = \kappa_2$$

We define a new widening operator $\bigtriangledown^\sharp$ in terms of the original abstract state widening operator $\bigtriangledown$ as follows:

$$\bigtriangledown^\sharp \in \mathcal{P}(\mathcal{P}(C)) \to \mathcal{P}(C)$$

$$\bigtriangledown^\sharp(S) = \left\{ \left\langle \bigtriangledown_{\langle \hat{\varsigma}, \kappa \rangle \in X} \hat{\varsigma}, \kappa \right\rangle \;\middle|\; X \in \left(\bigcup_{s \in S} s \right) \Big/ \sim \right\}$$

The input sets are unioned and the result is partitioned according to the equivalence relation $\sim$. For each partition the set of configurations are collapsed into a single configuration by joining together the abstract states (all configurations in the same partition are guaranteed to have the same knob value). The results over all partitions are unioned together, yielding a new set of abstract configurations.

Lemma 2. $\bigtriangledown^\sharp$ is a widening operator.

Proof. By definition of a widening operator. □

Now define the functor $^{\triangledown}$ so that for set S and function $\mathcal{F}$:

$$\overset{\triangledown}{\mathcal{F}}(S) = S \;\bigtriangledown^\sharp\; \mathcal{F}(S)$$

Then the set of reachable abstract configurations using widening over-approximates the set of reachable abstract configurations without widening—and thus over-approximates the set of reachable abstract states.

Theorem 2 (SOUNDNESS)**.**

$$\mathbf{lfp}^{\circ}_{\Sigma^\sharp_I} \mathcal{F}^\sharp \;\preceq\; \left\{ \pi_{\hat{\varsigma}}(c) \;\middle|\; c \in \mathbf{lfp}^{\circ}_{C_I} \overset{\triangledown}{\mathcal{G}} \right\}.$$

Proof. Follows from Lemmas 1 and 2. □

[1] Where $\pi_i(tuple)$ will denote the i^{th} component of *tuple*.

3.3 Control-Flow Sensitivity

The widening operator $\bigtriangledown^\sharp$ gives us the ability to tune the precision and performance of an analysis by defining the domain $\mathcal{K}$ and the transitions on that domain, independently from the rest of the analysis. However, the question remains: what are reasonable definitions to use in practice? We would like to lose as little precision as possible while remaining tractable; hence we should try to partition the states so that there are a tractable number of partitions *and* the states in each partition are as similar as possible.

A reasonable heuristic is to partition states based on how those states were computed—i.e., the execution history that led to each particular state. The hypothesis is that if two states were derived in a similar way then they are more likely to be similar. This is exactly the heuristic used by dataflow analysis when defining various *control-flow sensitivities*, such as flow-sensitive maximal fixed-point, *k*-CFA, object-sensitivity, property simulation, etc. They each compute some abstraction of the execution history (current program point, last *k* call-sites, last *k* allocation sites, etc.) and use that abstraction to partition the states during the analysis. We can formalize this insight by using a second-order abstract collecting semantics to derive the domain $\mathcal{K}$.

3.3.1 Second-Order Collecting Semantics

The first-order abstract collecting semantics loses all information about the relations between the states—i.e., the control-flow of the program as it executes. To abstract control-flow we need to retain this information, which leads us to *trace-based* abstract semantics and *second-order* collecting semantics. A *trace* $\tau \in \overrightarrow{\Sigma^\sharp}$ is a sequence of states $\langle \hat{\varsigma}_1, \hat{\varsigma}_2, \cdots, \hat{\varsigma}_n \rangle$ such that $\langle \hat{\varsigma}_i, \hat{\varsigma}_{i+1} \rangle \in \mathcal{F}^\sharp$. These traces define the control-flow of the program, therefore knowledge of these traces is essential for defining control-flow sensitivity. We can synthesize a trace-based semantics from the previous state-based semantics. Define the function $\mathcal{F}^\sharp_\uparrow$ so that for trace τ:

$$\mathcal{F}^\sharp_\uparrow(\tau) = \left\{ \tau :: \hat{\varsigma} \;\middle|\; \hat{\varsigma} \in \mathcal{F}^\sharp(last(\tau)) \right\}$$

where *last* returns the last state of a trace and the :: operator appends a state to the end of a trace. Then:

$$[\![P]\!]^\sharp_\uparrow = \mathbf{lfp}^{\circ}_{\Sigma^\sharp_i} \mathcal{F}^\sharp_\uparrow$$

This is a second-order collecting semantics: instead of all reachable abstract states, it yields all possible execution traces of abstract states. The first-order collecting semantics can be recovered from the second-order semantics: $[\![P]\!]^\sharp = \{\, last(\tau) \mid \tau \in [\![P]\!]^\sharp_\uparrow \,\}$.

With this formulation of the abstract semantics, analysis solutions are represented as sets of traces of abstract states. These traces give us the control-flow information we need, but not in a usable form due to the intractable size of the traces. To make the semantics tractable we must abstract these traces.

3.3.2 Abstracting Traces

Control-flow sensitivity either merges or keeps separate the abstract states in the analysis based on execution history, i.e., information contained in the traces that led to those particular states. However, different forms of control-flow sensitivity use different information contained in those traces. For example, a flow-sensitive analysis merges states that are produced at the same program point, so there is at most one abstract state per program point. A context-sensitive analysis further distinguishes control-flow information by the *context* in which a function was called, keeping abstract states arising from separate contexts apart. Different forms of context-sensitivity define "context" differently: for example, traditional *k*-CFA defines it as the last *k* call-sites encountered in the trace; stack-based *k*-CFA considers the top *k* currently active (i.e., not

yet returned) calls on the stack; object sensitivity considers abstract allocation sites instead of call-sites; and so on.

Thus we do not need to track the entire trace, we only need to track an abstraction of that trace which maintains the necessary information required by whatever form of control-flow sensitivity we wish to use. The particular trace abstraction $\Theta^\sharp$ is defined by the analysis designer to enforce the desired form of control-flow sensitivity:

$$\hat{\tau} \in \Theta^\sharp \qquad \text{abstract trace}$$
$$\mathcal{T}^\sharp \in \Sigma^\sharp \times \Sigma^\sharp \times \Theta^\sharp \to \Theta^\sharp \qquad \text{transition function}$$

The trace abstraction $\hat{\tau}$ contains the necessary control-flow information (e.g., current program point, last k call-sites, etc). The trace transition function $\mathcal{T}^\sharp$ produces an updated abstract trace reflecting the relevant control-flow information (e.g., new program point, new list of k call-sites, etc). Because $\mathcal{T}^\sharp$ tracks state transitions, it takes both the current and next states as arguments.

3.3.3 Tunable Control-Flow Sensitivity

To achieve a control-flow sensitive analysis we use the domain of abstract configurations with $\mathcal{K} = \Theta^\sharp$ and $\delta = \hat{\tau}$. The soundness result from Theorem 2 in Section 3.2 applies. The performance and precision of the resulting analysis is determined by the particular trace abstraction used; the coarser the trace abstraction the fewer the partitions, and thus the less precise the analysis.

By appropriately defining the abstract interpreter, the analysis designer can tune the control-flow sensitivity of the analysis without modifying the abstract semantics or interpreter implementation. Section 4 gives a detailed example of how to realize this framework for a specific language, semantics, and program analysis, including the specification of six different possible trace abstractions that yield known control-flow sensitivities from the dataflow analysis canon.

4. Example Analysis Framework

In this section we give a detailed example of how to use the framework established in Section 3 to build an abstract interpreter with tunable control-flow sensitivity. First we define an example language and give its concrete semantics, then we define the abstract semantics, and finally we show how to easily and modularly tune the control-flow sensitivity of the resulting abstract interpreter to achieve various levels of precision.

4.1 Syntax

Figure 1(a) gives the syntax of the example language we will analyze. It contains higher-order functions and mutable state. As discussed in Section 3.1, to specify control-flow sensitivity we must have an explicit representation of a program's control. We could achieve this in various ways, such as keeping the program in direct-style and using a continuation-passing denotational semantics or using a small-step abstract machine with explicit semantic continuations. For this example, we choose to syntactically transform programs into continuation-passing style (CPS) so that the control is explicit in the program syntax. Figure 1(b) gives the resulting CPS syntax—henceforth we assume that all programs are in this form. The CPS transformation is standard; we omit it for reasons of space.

As usual for CPS, expressions are separated into two categories: *Trivial* and *Serious* [23]. *Trivial* expressions are guaranteed to terminate and to have no side-effects; *Serious* expressions make no such guarantees. Functions can take an arbitrary number of arguments and can represent either user-defined functions from the direct-style program (modified to take an additional continuation parameter) or the continuations created by the CPS transform. We assume that it is possible to syntactically disambiguate among calls

$$n \in \mathbb{Z} \qquad x \in \mathit{Variable} \qquad \oplus \in \mathit{BinaryOp}$$

$$\begin{aligned} e \in \mathit{Exp} ::= {} & n \mid x \mid e_l \oplus e_r \mid x := e \mid e_1 \,;\, e_2 \mid \lambda\vec{x}.e \\ & \mid e_f(\vec{e}) \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{if}\ e_g\ e_t\ e_f \end{aligned}$$

(a) Direct-style abstract syntax.

$$n \in \mathbb{Z} \qquad x \in \mathit{Variable} \qquad \oplus \in \mathit{BinaryOp} \qquad \ell \in \mathit{Label}$$

$$L \in \mathit{Lam} ::= \lambda\vec{x}.\, S$$
$$T \in \mathit{Trivial} ::= n \mid x \mid L \mid T_l \oplus T_r$$
$$S \in \mathit{Serious} ::= \textbf{let}\ x = T\ \textbf{in}\ S \mid \textbf{set}\ x = T\ \textbf{in}\ S \mid \textbf{if}\ T\ S_t\ S_f \mid x(\vec{T})$$

(b) Continuation-passing style (CPS) abstract syntax.

Figure 1: Syntax for example language. Programs written in direct style are transformed to CPS, and the analysis proceeds over the CPS program.

to user-defined functions, calls to continuations that correspond to a function return, and all other calls. All syntactic entities have an associated unique label $\ell \in \mathit{Label}$; the expression $\cdot^\ell$ retrieves this label (for example, the label of *Serious* expression S is S^ℓ).

4.2 Concrete Semantics

We specify the concrete semantics of our language as a standard small-step abstract machine with environments and stores. Figure 2(a) gives the concrete semantic domains. A machine state consists of a *Serious* expression S, an environment ρ, and a store σ. An environment maps variables to addresses, and a store maps addresses to values. A value v can be either an integer or a closure. The language is dynamically typed, so that the same variable may—at different points during the computation—be bound to either an integer or a closure.

The semantic function η evaluates *Trivial* expressions. Its definition is given in Figure 2(b). A number requires no further evaluation, a variable is looked up using the environment and store, a lambda value evaluates to a closure, and a binary operation recursively evaluates its operands.

The concrete transition function $\mathcal{F}$ in Figure 2(c) transforms machine states; the definition is standard. Note that in the rule for function calls we use the notation $\overrightarrow{T}$ to mean the sequence of argument expressions, T_i to mean a particular argument expression, and $[\overrightarrow{p_i \to q_i}]$ to mean each p_i maps to its corresponding q_i.

The concrete first-order collecting semantics for our language is given by the least fixed-point of the semantic transition function $\mathcal{F}$, lifted to collect sets of states, starting from an initial set Σ_i:

$$[\![S]\!] = \mathbf{lfp}^{\circ}_{\Sigma_i} \mathcal{F}$$

This result is in general not computable because the *Value* domain is unbounded in both dimensions of the tuple, the integer domain and the closure domain (which is unbounded because of potentially infinitely many addresses). In the subsequent sections, we define a computable abstraction of these semantics with tunable control-flow sensitivity.

4.3 Abstract Semantics

This section presents a computable approximation of a program's behavior. We employ two specific data abstractions: for integers we use the constant propagation lattice $\mathbb{Z}^\sharp = \mathbb{Z} \cup \{\top_{\mathbb{Z}^\sharp}, \bot_{\mathbb{Z}^\sharp}\}$; for closures we use the powerset lattice of abstract closures. We use the

$$
\begin{array}{rcll}
\varsigma \in \Sigma & = & \mathit{Serious} \times \mathit{Env} \times \mathit{Store} & \text{(states)} \\
\rho \in \mathit{Env} & = & \mathit{Variable} \to \mathit{Address} & \text{(environments)} \\
\sigma \in \mathit{Store} & = & \mathit{Address} \to \mathit{Value} & \text{(stores)} \\
a \in \mathit{Address} & = & \mathbb{Z} & \text{(locations)} \\
\mathit{clo} \in \mathit{Closure} & = & \mathit{Env} \times \mathit{Lam} & \text{(closure values)} \\
v \in \mathit{Value} & = & \mathbb{Z} + \mathit{Closure} & \text{(program values)} \\
\mathcal{F} \in \Sigma \to \Sigma & = & \text{Figure 2c} & \text{(transition function)}
\end{array}
$$

(a) Concrete semantic domains.

$$\eta \in \mathit{Trivial} \times \mathit{Env} \times \mathit{Store} \to \mathit{Value}$$

$$\eta(n, \rho, \sigma) = n$$

$$\eta(x, \rho, \sigma) = \sigma(\rho(x))$$

$$\eta(\lambda\vec{x}\,.\,S, \rho, \sigma) = \langle \rho,\ \lambda\vec{x}\,.\,S \rangle$$

$$\eta(T_l \oplus T_r, \rho, \sigma) = \eta(T_l, \rho, \sigma) \oplus \eta(T_r, \rho, \sigma)$$

(b) Concrete *Trivial* evaluation.

S	*conditions*	S'	ρ'	σ'
let $x = T$ **in** S_b	$[\![T]\!] = v$	S_b	$\rho[x \mapsto a']$	$\sigma[a' \mapsto v]$
set $x = T$ **in** S_b	$[\![T]\!] = v \wedge \rho(x) = a$	S_b	ρ	$\sigma[a \mapsto v]$
if T S_t S_f	$[\![T]\!] \neq 0$	S_t	ρ	σ
if T S_t S_f	$[\![T]\!] = 0$	S_f	ρ	σ
$x(\overrightarrow{T})$	$[\![T_i]\!] = v_i \wedge [\![x]\!] = \langle \rho_c,\ \lambda\vec{y}\,.\,S_b \rangle$	S_b	$\rho_c[\overrightarrow{y_i \mapsto a'_i}]$	$\sigma[\overrightarrow{a'_i \mapsto v_i}]$

(c) Concrete transition function $\mathcal{F}$, where $[\![\cdot]\!] = \eta(\cdot, \rho, \sigma)$ and a' is a fresh address. Given a current state $\varsigma = \langle S, \rho, \sigma \rangle$, the transition function yields a new state $\mathcal{F}(\varsigma) = \langle S', \rho', \sigma' \rangle$.

Figure 2: Concrete semantics. The semantic transition function $\mathcal{F}$ over concrete states Σ (Figure 2a) is formulated as a small-step abstract machine with separate evaluation for *Trivial* (Figure 2b) and *Serious* expressions (Figure 2c).

convention from abstract interpretation that $\bot$ is the most precise value and $\top$ is the least precise value in a lattice.

The analysis employs a finite abstract address domain *Address* and a function *alloc* that generates new abstract addresses. The definitions of *Address* and *alloc* control the *heap sensitivity* of the analysis—in other words, the granularity with which the analysis divides memory into a finite number of partitions. The more fine-grained this partitioning the more precise the analysis, but also the more expensive it may become. The heap sensitivity of the analysis is tangentially related to control-flow sensitivity; we leave these definitions unspecified in this section, but discuss the issue further in Section 4.4.

Figure 3(a) gives the abstract semantic domains. The main differences with the concrete semantic domains are (1) all of the domains are finite; (2) the $\mathit{Value}^\sharp$ domain is a pair of values rather than a single type of value; and (3) some of the domains have been lifted to sets. An abstract value is a pair because the language is not statically typed and thus we cannot restrict the potential values a variable can have. Domains have been lifted to sets because abstract states form a lattice and any two states must have a join.

The abstract *Trivial* evaluation is standard; its definition is given in Figure 3(b). The main differences from concrete trivial evaluation are that variable lookup joins all the abstract values to which the variable may be bound and that abstract evaluation for integers and closures yield a product.

The abstract semantic transition function $\mathcal{F}^\sharp$ transforms abstract states. Figure 3(c) gives its definition. The main feature of note is that the analysis employs *weak updates*, i.e., when a value is updated, the analysis joins the new value with the old value. It is possible under certain circumstances to strongly update the store (by replacing the old value instead of joining with it), but for simplicity our example uses weak updates.

The abstract first-order collecting semantics for our language is given by the least fixed-point of the transition function $\mathcal{F}^\sharp$, lifted to collect sets of abstract states, starting from an initial set $\Sigma_i^\sharp$:

$$[\![S]\!]^\sharp = \mathbf{lfp}^{\circ}_{\Sigma_i^\sharp} \mathcal{F}^\sharp$$

This analysis is sound and decidable. However, it is still intractable—the set of reachable states grows exponentially with the number of nondeterministic choices. To make this analysis tractable requires some form of control-flow sensitivity.

4.4 Control-Flow Sensitivity

We now extend the abstract semantics for our language to express tunable control-flow sensitivity. As described in Section 3, we extend the abstract semantics to include a "knob" domain that corresponds to trace abstractions. We design the semantics so that it is parameterized by the definition of the trace abstraction, thus allowing the control-flow sensitivity of the analysis to be modified by plugging in different trace abstractions. The parameters are the trace abstraction domain $\Theta^\sharp$ and the trace update function $\mathcal{T}^\sharp$.

The most general way to parameterize an existing abstract semantics is to (1) define abstract configurations as the cross-product of the abstract state and abstract trace domains and (2) to define an abstract transition function $\mathcal{G}$ that applies the original abstract semantic transition function $\mathcal{F}^\sharp$ and separately applies the trace transition function $\mathcal{T}^\sharp$. In this way, we can add control-flow sensitivity without modifying the existing abstract semantics.

Although this approach is the most general, it may not be the most efficient way to compute control-flow sensitivity for a specific abstract semantics. For example, it may often be the case that $\mathcal{T}^\sharp$ needs a subset of the information computed by $\mathcal{F}^\sharp$ (e.g., it may need to know the next program point). Rather than have $\mathcal{T}^\sharp$ repeat the work of $\mathcal{F}^\sharp$, we can slightly modify the abstract semantics to incorporate traces without sacrificing tunability.

In particular, we make three changes to the abstract semantics of the previous section to integrate trace abstractions into the semantics. First we add traces directly to the abstract state. This change is cosmetic—it flattens the cross-product of two domains into a single domain. Next we modify $\mathcal{F}^\sharp$ to operate directly on this flattened domain. This change gives the trace update mechanism access to all the data needed to compute a new abstract trace, which avoids redundant work. The final change extends abstract closures to contain an abstract trace. Intuitively, a closure's abstract trace corresponds to the trace that existed before a function was called. Any analysis that tracks calls and returns (e.g., stack-based k-CFA) can use this extra information to simulate stack behavior upon exiting a function call by restoring the trace to the point before a function was called.

$\hat{n} \in \mathbb{Z}^\sharp \qquad \hat{\oplus} \in \mathit{BinaryOp}^\sharp \qquad \hat{a} \in \mathit{Address}^\sharp$

$$
\begin{array}{rcll}
\hat{\varsigma} \in \Sigma^\sharp & = & \mathcal{P}(\mathit{Serious}) \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp & \text{(abstract states)} \\
\hat{\rho} \in \mathit{Env}^\sharp & = & \mathit{Variable} \to \mathcal{P}(\mathit{Address}^\sharp) & \text{(environments)} \\
\hat{\sigma} \in \mathit{Store}^\sharp & = & \mathit{Address}^\sharp \to \mathit{Value}^\sharp & \text{(stores)} \\
\widehat{clo} \in \mathit{Closure}^\sharp & = & \mathcal{P}(\mathit{Env}^\sharp \times \mathit{Lam}) & \text{(closure values)} \\
\hat{v} \in \mathit{Value}^\sharp & = & \mathbb{Z}^\sharp \times \mathit{Closure}^\sharp & \text{(abstract values)} \\
\mathcal{F}^\sharp \in \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp) & = & \text{Figure 3c} & \text{(transition function)}
\end{array}
$$

(a) Abstract semantic domains.

$$\hat{\eta} \in \mathit{Trivial} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \to \mathit{Value}^\sharp$$

$$
\begin{aligned}
\hat{\eta}(n, \hat{\rho}, \hat{\sigma}) &= \langle \hat{n},\ \emptyset \rangle \\
\hat{\eta}(x, \hat{\rho}, \hat{\sigma}) &= \bigsqcup_{\hat{a} \in \hat{\rho}(x)} \hat{\sigma}(\hat{a}) \\
\hat{\eta}(\lambda \vec{x}.\, S, \hat{\rho}, \hat{\sigma}) &= \langle \bot_{\mathbb{Z}^\sharp},\ \{\langle \hat{\rho},\ \lambda \vec{x}.\, S \rangle\} \rangle \\
\hat{\eta}(T_l \oplus T_r, \hat{\rho}, \hat{\sigma}) &= \hat{\eta}(T_l, \hat{\rho}, \hat{\sigma}) \ \hat{\oplus}\ \hat{\eta}(T_r, \hat{\rho}, \hat{\sigma})
\end{aligned}
$$

(b) Abstract *Trivial* evaluation.

$S_i \in \overline{S}$	*conditions*	S'	$\hat{\rho}'$	$\hat{\sigma}'$
let $x = T$ **in** S_b	$[\![T]\!] = \hat{v}$	S_b	$\hat{\rho}[x \mapsto \hat{a}']$	$\hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}]$
set $x = T$ **in** S_b	$[\![T]\!] = \hat{v} \wedge \hat{\rho}(x) = \overrightarrow{\hat{a}}$	S_b	$\hat{\rho}$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}_i \mapsto \hat{v}}]$
if T S_t S_f	$\pi_{\hat{n}}([\![T]\!]) \notin \{\hat{0}, \bot_{\mathbb{Z}^\sharp}\}$	S_t	$\hat{\rho}$	$\hat{\sigma}$
if T S_t S_f	$\pi_{\hat{n}}([\![T]\!]) \sqsupseteq \hat{0}$	S_f	$\hat{\rho}$	$\hat{\sigma}$
$x(\overrightarrow{T})$	$[\![T_i]\!] = \hat{v}_i \ \wedge\ \pi_{\widehat{clo}}([\![x]\!]) \ni \langle \hat{\rho}_c,\ \lambda \vec{y}.\, S_c \rangle$	S_c	$\hat{\rho}_c[\overrightarrow{y_i \mapsto \hat{a}'_i}]$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}'_i \mapsto \hat{v}_i}]$

(c) Abstract transition function $\mathcal{F}^\sharp$, where $[\![\cdot]\!] = \hat{\eta}(\cdot, \hat{\rho}, \hat{\sigma})$ and $\hat{a}'$ is given by *alloc*. Given a current state $\hat{\varsigma} = \langle \overline{S}, \hat{\rho}, \hat{\sigma} \rangle$, the transition function yields a set of new states $\mathcal{F}^\sharp(\hat{\varsigma}) = \overline{\langle \{S'\}, \hat{\rho}', \hat{\sigma}' \rangle}$.

Figure 3: Abstract semantics. The semantic transition function $\mathcal{F}^\sharp$ over abstract states $\Sigma^\sharp$ (Figure 3a) is formulated as a small-step abstract machine with separate evaluation for *Trivial* (Figure 3b) and *Serious* expressions (Figure 3c).

$\hat{n} \in \mathbb{Z}^\sharp \qquad \hat{\oplus} \in \mathit{BinaryOp}^\sharp \qquad \boxed{\hat{\tau} \in \mathit{Trace}^\sharp}$

$$
\begin{array}{rcll}
\hat{\varsigma} \in \Sigma^\sharp & = & \mathit{Serious}^\sharp \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \boxed{\mathit{Trace}^\sharp} & \text{(abstract states)} \\
\hat{\rho} \in \mathit{Env}^\sharp & = & \mathit{Variable} \to \mathcal{P}(\mathit{Address}^\sharp) & \text{(environments)} \\
\hat{\sigma} \in \mathit{Store}^\sharp & = & \mathit{Address}^\sharp \to \mathit{Value}^\sharp & \text{(stores)} \\
\hat{a} \in \mathit{Address}^\sharp & = & \mathit{Label} \times \mathit{Trace}^\sharp & \text{(locations)} \\
\widehat{clo} \in \mathit{Closure}^\sharp & = & \mathcal{P}(\boxed{\mathit{Trace}^\sharp} \times \mathit{Env}^\sharp \times \mathit{Lam}) & \text{(closure values)} \\
\hat{v} \in \mathit{Value}^\sharp & = & \mathbb{Z}^\sharp \times \mathit{Closure}^\sharp & \text{(abstract values)} \\
\mathcal{G} \in \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp) & = & \text{Figure 4c} & \text{(transition function)}
\end{array}
$$

(a) Abstract semantic domains.

$$\hat{\eta} \in \mathit{Trivial} \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \boxed{\mathit{Trace}^\sharp} \to \mathit{Value}^\sharp$$

$$
\begin{aligned}
\hat{\eta}(n, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \hat{n},\ \{\} \rangle \\
\hat{\eta}(x, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \bigsqcup_{\hat{a} \in \hat{\rho}(x)} \hat{\sigma}(\hat{a}) \\
\hat{\eta}(\lambda \vec{x}.\, S, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \bot_{\mathbb{Z}^\sharp},\ \{\langle \boxed{\hat{\tau}},\ \hat{\rho},\ \lambda \vec{x}.\, S \rangle\} \rangle \\
\hat{\eta}(T_l \oplus T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \hat{\eta}(T_l, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \ \hat{\oplus}\ \hat{\eta}(T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau})
\end{aligned}
$$

(b) Abstract *Trivial* evaluation.

$S_i \in \overline{S}$	*conditions*	S'	$\hat{\rho}'$	$\hat{\sigma}'$	$\boxed{\hat{\tau}'}$
let $x = T$ **in** S_b	$[\![T]\!] = \hat{v}$	S_b	$\hat{\rho}[x \mapsto \hat{a}']$	$\hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{v}]$	$\tau_{stmt}(\hat{\varsigma}, S_b)$
set $x = T$ **in** S_b	$[\![T]\!] = \hat{v} \wedge \hat{\rho}(x) = \overrightarrow{\hat{a}}$	S_b	$\hat{\rho}$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}_i \mapsto \hat{v}}]$	$\tau_{stmt}(\hat{\varsigma}, S_b)$
if T S_t S_f	$\pi_{\hat{n}}([\![T]\!]) \notin \{\hat{0}, \bot_{\mathbb{Z}^\sharp}\}$	S_t	$\hat{\rho}$	$\hat{\sigma}$	$\tau_{stmt}(\hat{\varsigma}, S_t)$
if T S_t S_f	$\pi_{\hat{n}}([\![T]\!]) \sqsupseteq \hat{0}$	S_f	$\hat{\rho}$	$\hat{\sigma}$	$\tau_{stmt}(\hat{\varsigma}, S_f)$
$x(\overrightarrow{T})$	$[\![T_i]\!] = \hat{v}_i \ \wedge\ \pi_{\widehat{clo}}([\![x]\!]) \ni \langle \boxed{\hat{\tau}_c},\ \hat{\rho}_c,\ \lambda \vec{y}.\, S_c \rangle$	S_c	$\hat{\rho}_c[\overrightarrow{y_i \mapsto \hat{a}'_i}]$	$\hat{\sigma} \sqcup [\overrightarrow{\hat{a}'_i \mapsto \hat{v}_i}]$	$\tau_{call}(\hat{\varsigma}, \langle \hat{\tau}_c, \hat{\rho}_c, \lambda \vec{y}.\, S_c \rangle)$

(c) Abstract semantic function $\mathcal{G}$, where $[\![\cdot]\!] = \hat{\eta}(\cdot, \hat{\rho}, \hat{\sigma}, \hat{\tau})$ and $\hat{a}'$ is given by *alloc*. Given a current state $\hat{\varsigma} = \langle \overline{S}, \hat{\rho}, \hat{\sigma}, \hat{\tau} \rangle$, the transition function yields a set of new states $\mathcal{G}(\hat{\varsigma}) = \overline{\langle \{S'\}, \hat{\rho}', \hat{\sigma}', \tau' \rangle}$.

Figure 4: Abstract semantics, specialized for tunable control-flow sensitivity. The boxed elements highlight the specialization.

We note that these changes do not affect the generality or tunability of our framework; it is possible to express control-flow sensitivity for our example in a separable way. However, these changes make it simpler to specify tunable control-flow sensitivity for our example abstract machine.

Figure 4 defines our example analysis, extended to incorporate tunable control-flow sensitivity. The boxed elements in this diagram highlight the differences between this semantics and the abstract semantics of the previous section. Abstract states and abstract closures have been extended to include an abstract trace (Figure 4a). Note that we give a specific finite domain for abstract addresses: $\mathit{Label} \times \mathit{Trace}^\sharp$. This definition allows the analysis designer to choose a heap sensitivity that uses the maximum amount of precision available (i.e., the entire abstract trace) or some truncated version, thereby reducing precision but possibly increasing performance. Trivial evaluation of a lambda expression stores the current trace in its closure (Figure 4b). The transition function uses tunable control-flow sensitivity to update the trace (Figure 4c).

The analysis designer tunes control-flow sensitivity by specifying an abstract trace domain and a pair of transition functions that generate new abstract traces:

$$\hat{\tau} \in \mathit{Trace}^\sharp$$
$$\tau_{\mathit{stmt}} \in \Sigma^\sharp \times \mathit{Serious} \to \mathit{Trace}^\sharp$$
$$\tau_{\mathit{call}} \in \Sigma^\sharp \times \mathit{Closure}^\sharp \to \mathit{Trace}^\sharp$$

The abstract trace domain summarizes the history of program execution. The abstract trace transition function τ_{stmt} specifies how to generate a trace when execution transitions between two program points in the same function. The abstract trace transition function τ_{call} specifies how to generate a trace when execution transitions across a function call.

Starting from an initial set $\Sigma_i^\sharp$, the collecting semantics for this analysis is given by the least fixed-point of the transition function $\mathcal{G}$, lifted to collect sets of abstract states and widened at each step to join states that contain the same abstract trace:

$$[\![S]\!]^\sharp = \mathbf{lfp}^{\triangledown}_{\Sigma_i^\sharp} \hat{\mathcal{G}}$$

The tractability of this analysis depends on the particular choice for control-flow sensitivity. In the remainder of this section we tune the analysis to six different forms of control-flow sensitivity. We could define many more, but our choices are sufficient to demonstrate the utility and flexibility of our framework.

4.4.1 Flow-insensitive, context-insensitive analysis

In a flow-insensitive analysis any *Serious* expression can execute after any other *Serious* expression, regardless of where those expressions appear in the program. Rather than compute separate solutions for each program point, the analysis computes a single solution for the entire program. In our framework, the analysis designer can specify flow-insensitive analysis by making the $\mathit{Trace}^\sharp$ domain a single value, so that all states will necessarily have the same abstract trace.

Analysis 1 Flow-insensitive, context-insensitive

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathbf{1}$$
$$\tau_{\mathit{stmt}}(\_, \_) = \mathbf{1}$$
$$\tau_{\mathit{call}}(\_, \_) = \mathbf{1}$$

The abstract semantics joins every state into one single state, i.e., the fixed-point computation continually adds information to a single program-wide state until that state converges.

This analysis, as specified, is tractable (i.e., it has polynomial complexity) but somewhat inefficient because of the way the analysis processes states that contain multiple expressions. Effectively, the analysis uses a round-robin strategy so that if the solution changes after any one *Serious* expression, all *Serious* expressions in that state are reevaluated. Part of our future work is to extend our framework to optimize this fixed-point computation without forcing the analysis designer to bake in their own fixed-point algorithm.

4.4.2 Flow-sensitive (FS), context-insensitive analysis

A flow-sensitive analysis executes statements in program-order, computing a single solution for each program point. The analysis designer can specify flow-sensitive analysis by making the $\mathit{Trace}^\sharp$ domain the set of program labels and updating the trace at each step to be the current program point.

Analysis 2 Flow-sensitive, context-insensitive

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathit{Label}$$
$$\tau_{\mathit{stmt}}(\_, S) = S^\ell$$
$$\tau_{\mathit{call}}(\_, \langle \_, \_, \lambda \vec{y}\,.\, S_c \rangle) = S_c^\ell$$

The abstract semantics at each step collects all states at the same program point and join them together, constraining the maximum number of abstract states to be the number of program points.

4.4.3 FS + traditional k-CFA analysis

Traditional k-CFA [26] is a context-sensitive analysis that keeps track of the last k call-sites encountered along an execution path and uses this *callstring* to distinguish information at a given program point that arrives via different routes. At each function call the analysis appends the call-site to the callstring and truncates the result so that the new callstring has at most k elements. Within a function, the analysis can be flow-insensitive or flow-sensitive—flow-sensitivity makes the most sense and matches the behavior achieved by converting **let** and **set** into calls. The analysis designer can specify flow-sensitive k-CFA by making the $\mathit{Trace}^\sharp$ domain contain a tuple of the current program point and the callstring (as a sequence of labels). The first element of the tuple tracks flow-sensitivity; the second element of the tuple tracks context-sensitivity.

Analysis 3 Flow-sensitive, k-CFA

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathit{Label} \times \mathit{Label}^\star$$
$$\tau_{\mathit{stmt}}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$
$$\tau_{\mathit{call}}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \_, \_, \lambda \vec{y}\,.\, S_c \rangle) = \langle S_c^\ell,\ \hat{\tau}' \rangle$$
$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

The τ_{stmt} transition function is the same as for flow-sensitivity. The τ_{call} transition function distinguishes between user-defined calls and continuation calls that were introduced during the CPS transformation. For a user-defined call, the transition function updates the callstring; for continuations, it leaves the callstring as-is.

Note that the current callstring is left unmodified when returning from a call (i.e., calling the continuation that was passed into the current function); thus the callstring does *not* act like a stack. This means that two execution paths that entered a function with different callstrings could, depending on the calls that they make

inside the function, end up with the same callstring at the same program point somewhere inside the function.

4.4.4 FS + stack-based *k*-CFA analysis

In dataflow analysis *k*-CFA is usually defined as having a stack-like behavior, so that upon returning from a function call the current callstring is discarded and replaced by the callstring that held immediately before making that function call (in effect, the callstring is "pushed" when entering a function and "popped" when exiting the function). The analysis designer can achieve this behavior by modifying τ_{call} to detect continuation calls that correspond to function returns and to replace the current callstring with the callstring held in the return continuation's closure. The CPS transformation guarantees this callstring to be the one that held immediately before the current function was called.

Analysis 4 Flow-sensitive, stack-based *k*-CFA

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathit{Label} \times \mathit{Label}^\star$$
$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$
$$\tau_{call}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \hat{\tau}_c, \_, \lambda \vec{y}\,.\, S_c \rangle) = \langle S_c^\ell,\ \hat{\tau}' \rangle$$
$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

Since an execution path is guaranteed to have the same callstring inside a function as the one it entered with, no matter what calls it makes inside the function, this analysis will not join two states with differing environments; hence the domain of any abstract environment will always be a singleton set. In addition, it now makes more sense than traditional *k*-CFA to use flow-insensitive analysis within a function (though we still specify flow-sensitive analysis above).

4.4.5 FS + *k*-allocation-site sensitive analysis

Object-sensitivity [19] is a popular form of context-sensitive control-flow sensitivity for object-oriented languages. We do not have objects in our example language, but as noted elsewhere [27] object-sensitivity should more properly be termed *allocation-site* sensitivity—it defines a function's context in terms of the last *k* allocation-sites (i.e., abstract addresses) rather than callstrings. Under the assumption that every function call uses a variable as the first argument, the analysis designer can employ a form of allocation-site sensitivity by using that variable's address to form the trace.

Analysis 5 Flow-sensitive, *k*-allocation-site sensitive

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathit{Label} \times \mathit{Address}^{\sharp\star}$$
$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$
$$\tau_{call}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \hat{\tau}_c, \_, \lambda \vec{y}\,.\, S_c \rangle) = \langle S_c^\ell,\ \hat{\tau}' \rangle$$
$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (\textbf{self} :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

The value **self** refers to the address of the call's first argument. In an object-oriented language, this argument always corresponds to the receiver of a method (i.e., the **self** or **this** pointer).

4.4.6 Property simulation analysis

A more unusual form of control-flow sensitivity is Das et al.'s property simulation [8]. The previous sensitivities we have described use low-level notions of execution trace, either in terms of calls or addresses. Property simulation relies on a finite-state machine (FSM) that describes a higher-level notion of execution trace—for example, an FSM whose states track whether a file is open or closed, or whether a lock is locked or unlocked. The analysis transitions this FSM according to the instructions it encounters. At a join point in the program (e.g., immediately after the two branches of a conditional) the analysis either merges the execution state or not depending on whether the FSMs along the two paths are in the same FSM state. The analysis designer can specify property simulation by making $\mathit{Trace}^\sharp$ be a tuple that contains the current program point and the current state of the FSM. The FSM is updated based on an API (e.g., for file or lock operations) so that τ_{call} will transition the FSM accordingly.

Analysis 6 Flow-sensitive, property-sensitive

$$\hat{\tau} \in \mathit{Trace}^\sharp = \mathit{Label} \times \mathit{FSM}$$
$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$
$$\tau_{call}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \hat{\tau}_c, \_, \lambda \vec{y}\,.\, S_c \rangle) = \langle S_c^\ell,\ \delta_{FSM}(S, \pi_2(\hat{\tau})) \rangle$$

5. JavaScript Analysis and Evaluation

In this section we demonstrate that our method scales to real-world languages by implementing and evaluating a tunable abstract interpreter for JavaScript. The major language features of JavaScript are mutability, higher-order functions, objects, prototype-based inheritance, and exceptions. It is dynamically typed, object fields can be added and deleted arbitrarily at any time, and it uses computable field accesses (i.e., the object fields being accessed and updated can be specified using arbitrary language expressions). JavaScript is notoriously resistant to static analysis; it has many obscure and surprising corner cases that are not formally specified nor well-documented. Creating an analysis that is both sound and precise is a difficult problem. To help deal with these difficulties we create a formally-specified core JavaScript calculus called notJS; we then desugar JavaScript programs to this core calculus in the spirit of Guha et al's λ_{JS} [9]. This core calculus greatly simplifies the cases we need to reason about and allows us to formally reason about the correctness of our analysis.

5.1 The notJS Core Calculus

The notJS language has a smallstep operational semantics and a *desugar* transformation that translates JavaScript into notJS. The desugaring translation and the notJS semantics have been rigorously tested against the actual behavior of JavaScript programs using the Mozilla JavaScript implementation. The language semantics, desugarer, concrete interpreter, and abstract interpreter are open source and freely available for download.

Figure 5 gives the notJS abstract syntax and Figure 6(a) gives the concrete semantic domains. The semantics contains 79 rules and 13 semantic helper functions which are omitted for space, but are available for download with the rest of the notJS language.[2]

5.2 The notJS Abstract Interpreter

Our JavaScript analysis simultaneously performs control-flow analysis (for each call-site, which functions/methods may be called), pointer analysis (for each object reference, which objects may be

[2] Note to reviewer: available for review by contacting the program chair.

$$\varsigma \in State = (Value + Exp) \times Env \times Store \times Kont$$
$$\rho \in Env = Variable \to Address$$
$$\sigma \in Store = Address \to (BaseValue + Object)$$
$$a \in Address = \mathbb{Z}$$
$$clo \in Closure = Env \times Lam$$
$$bv \in BaseValue = \mathbb{Z} + \mathbb{B} + String + Address + Closure + Null + Undef$$
$$obj \in Object = String \to BaseValue$$
$$jv \in JmpValue = Label \times Value$$
$$v \in Value = BaseValue + JmpValue$$
$$\kappa \in Kont = \ldots \text{elided for space}$$

(a) Concrete semantic domains.

$$\hat{\varsigma} \in State^\sharp = (Value^\sharp + Exp) \times Env^\sharp \times Store^\sharp \times Trace^\sharp \times Kont^\sharp$$
$$\hat{\rho} \in Env^\sharp = Variable \to \mathcal{P}(Address^\sharp)$$
$$\hat{\sigma} \in Store^\sharp = Address^\sharp \to (BaseValue^\sharp + Object^\sharp)$$
$$\hat{a} \in Address^\sharp = Label \times Trace^\sharp$$
$$\widehat{clo} \in Closure^\sharp = \mathcal{P}(Env^\sharp \times Lam)$$
$$\widehat{bv} \in BaseValue^\sharp = \mathbb{Z}^\sharp \times \mathbb{B}^\sharp \times String^\sharp \times \mathcal{P}(Address^\sharp) \times Closure^\sharp \times Null^\sharp \times Undef^\sharp$$
$$\widehat{obj} \in Object^\sharp = String^\sharp \to BaseValue^\sharp$$
$$\widehat{jv} \in JmpValue^\sharp = Label \times Value^\sharp$$
$$\hat{v} \in Value^\sharp = BaseValue^\sharp + JmpValue^\sharp$$
$$\hat{\kappa} \in Kont^\sharp = \ldots \text{elided for space}$$

(b) Abstract semantic domains.

Figure 6: Concrete and abstract semantic domains for notJS. Note that the abstract trace domain $Trace^\sharp$ is intentionally left unspecified.

$$n \in \mathbb{Z} \quad b \in \mathbb{B} \quad str \in String \quad x \in Variable \quad \ell \in Label$$

$$\begin{aligned}
L \in Lam &::= (\texttt{self}, \vec{x}) \to e \\
e \in Exp &::= n \mid b \mid str \mid x \mid \textbf{null} \mid \textbf{undef} \mid L \mid e_1; e_2 \\
&\mid \odot\, e \mid e_1 \oplus e_2 \mid e_1(\vec{e_2}) \mid \textbf{var}\ \vec{x}\ \textbf{in}\ e \mid x := e \\
&\mid \textbf{if}\ e_1\ e_2\ \textbf{else}\ e_3 \mid \textbf{while}\ e_1\ e_2 \mid \ell\ e \mid \textbf{brk}\ \ell\ e \\
&\mid \textbf{try}\ e_1\ \textbf{catch}\ x\ e_2\ \textbf{finally}\ e_3 \mid \textbf{throw}\ e \mid \textbf{eval}\ e \\
&\mid \left\{\overrightarrow{str : e}\right\} \mid e_1.e_2 \mid e_1.e_2 := e_3 \mid \textbf{del}\ e \\
\oplus \in BinOp &::= + \mid - \mid \times \mid \div \mid \wedge \mid \vee \mid = \mid \leq \mid << \mid >> \mid >>> \\
&\mid \textbf{xor} \mid \textbf{band} \mid \textbf{bor} \mid \textbf{charAt} \mid \textbf{hasOwnProperty} \\
\odot \in UnOp &::= \textbf{bnot} \mid \textbf{floor} \mid \neg \mid \textbf{length} \mid \textbf{keys} \\
&\mid \textbf{typeof} \mid \textbf{numToString} \mid \textbf{stringToNum}
\end{aligned}$$

Figure 5: Direct-style notJS abstract syntax.

accessed), type inference (for each value, can it be a number, a boolean, a string, `null`, `undef`, a closure, or an object), and integer, boolean, and string constant propagation (for each such value, is it a known constant value). These analyses are all complementary in the sense that the information from each one can help improve the precision of the others. For example, string constant propagation is critical to determine which fields and methods of an object may be accessed or updated, while boolean constant propagation can help eliminate infeasible control-flow paths.

Figure 6(b) gives the analysis abstract domains. An abstract value is a tuple of all the relevant abstract types; this is because JavaScript is dynamically typed and hence we cannot restrict expressions to have only a single kind of value. Each abstract type is a lattice: `null` and `undef` use two-level lattices indicating that a value either definitely is *not* of that type (the $\bot$ value) or that it may be of that type (the $\top$ value). Numbers, booleans and strings each use the standard constant propagation lattice; closures use a powerset lattice of *(function, environment)* pairs; and objects use a function lattice containing maps from abstract strings to abstract values. We use the abstract interpretation convention that lattice bottom is the most precise value and lattice top is the least precise value.

5.3 Evaluation

To evaluate our framework, we implement our parameterized abstract interpreter in Scala and run it on a suite of JavaScript programs. Our intent is to show that our technique is scalable to a real-world language and that it yields a tractable abstract interpreter that can be tuned to various degrees of control-flow sensitivity. As such, we report only performance results for these analyses. Although a comparison of their relative precision would also be interesting, defining a precision metric that can provide meaningful comparisons across all of the different forms of sensitivity is a difficult task in itself and is left for future work.

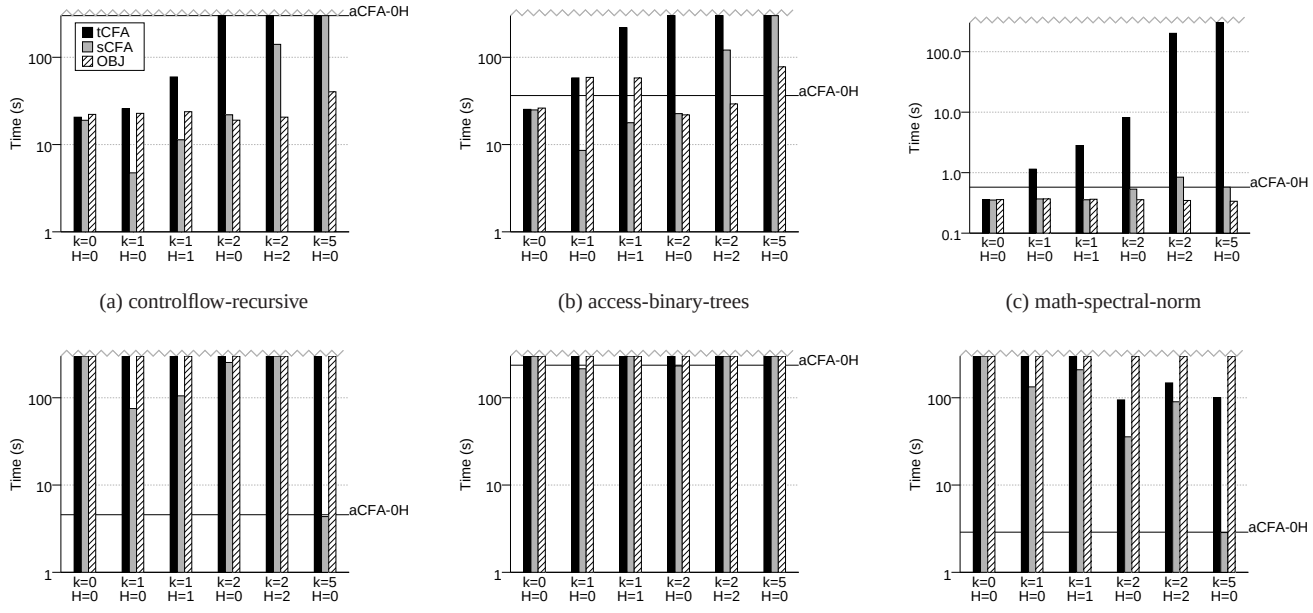
Benchmarks and methodology. We evaluate our framework using the SunSpider suite of JavaScript benchmarks, version 0.9.1.[3] This suite is designed to include real-world JavaScript programs that collectively employ a broad range of language features. We omit from our study three benchmarks that use JavaScript's **eval**, the analysis of which is outside the scope of this work. The remaining 23 benchmarks range in size from 3 to 1,696 lines of original JavaScript code and from 98 to 10,035 notJS AST nodes. We execute the benchmarks on a 3.2 GHz Xeon processor with 6GB of memory, running Scala 2.9.1. Each benchmark executes several times to warm up the JVM and achieve a steady-state for the JIT compiler, after which we measure a run with a five minute timeout.[4] We report detailed results for six of the most interesting and difficult-to-analyze benchmarks.

Analyses. We instantiate 19 analyses in our framework, covering a range of control-flow sensitivities. Each analysis is flow-sensitive because that is the minimum level of precision required to get useful information from a JavaScript analysis. The analyses cover four families of context-sensitivity: traditional *k*-CFA, stack-based *k*-CFA, *k*-object-sensitive, and acyclic callstrings[5] [14]. For the acyclic callstrings family, we instantiate a single member which is heap-insensitive (acfa-0H). For each of the other families, we instantiate six members by varying the value of *k* and the heap-sensitivity *h*. The template for specifying these members is: *k*-{tcfa, scfa, obj}-*h*H, where tcfa is traditional *k*-CFA, scfa is stack-based

[3] `www.webkit.org/perf/sunspider/sunspider.html`

[4] We use Brent Boyer's benchmarking system to control our evaluation. `www.ellipticgroup.com/html/benchmarkingArticle.html`

[5] A callstring-based abstraction that collapses cycles into a single point.



(a) controlflow-recursive (b) access-binary-trees (c) math-spectral-norm

(d) crypto-md5 (e) 3d-cube (f) 3d-raytrace

Figure 7: Execution times. The ACFA analysis is represented as horizontal black line; the jagged line at top represents timeout.

k-CFA, and OBJ is *k*-object-sensitivity. Our framework requires no more than 15 lines of Scala code to implement an entire family of analyses (e.g., TCFA, SCFA, OBJ, or ACFA).

Results. Figure 7 contains detailed results for six benchmarks; we highlight several interesting observations. Our results demonstrate the utility of tunable control-flow sensitivity. No clear winner emerges within or across families of analyses—instead, the analysis designer should evaluate a variety of sensitivities and tune the analysis to meet their needs.

Object sensitivity is a popular control-flow sensitivity to employ for object-oriented languages. Our results demonstrate that for typical values of *k* (i.e., 0 and 1), stack-based *k*-CFA performs as well or better than *k*-object-sensitivity for these JavaScript programs. Although these results do not provide complete insight into the relation between these two families of context-sensitivity, they indicate that the community could benefit from a more detailed study on the choice of context abstraction for JavaScript.

Many people mistakenly expect that execution time increases with analysis precision, but our results show that the relation between the two is more complex. For example, the cost of traditional *k*-CFA increases exponentially with *k* for math-spectral-norm but not for 3d-raytrace. More surprisingly, context-insensitive stack-based CFA (i.e., 0-sCFA-0H) times out for crypto-md5 but finishes in under 5 seconds for $k = 5$.

6. Related Work

Abstract interpretation has a broad and deep body of research. However, there is relatively little work on abstract interpretation with control-flow sensitivity. None of that work has couched control-flow sensitivity in terms of a widening operator based on abstractions of the second-order collecting semantics, and none of that work allows for tractable, tunable control-flow sensitivity.

Trace Partitioning. Our work is similar in some respects to the trace partitioning work by Mauborgne and Rival [16, 24]. Trace partitioning was developed in the context of the ASTRÉE static analyzer [7] for a restricted subset of the C language, primarily intended for embedded systems. Mauborgne and Rival observe that usually abstract interpreters are (1) based on first-order collecting semantics, making it difficult to express control-flow sensitivity; and (2) designed to silently merge information at control-flow join points[6]—what in dataflow analysis is called "flow-sensitive maximal fixed-point". They propose a method to postpone these silent merges when doing so can increase precision; effectively they are adding a controlled form of path-sensitivity to the analysis.

Mauborgne and Rival describe a denotational semantics-based analysis that can use three different criteria to determine whether to merge information at a particular point: the last *k* branch decisions taken (i.e., whether an execution path took the *true* or *false* branch); the last *k* while-loop iterations (effectively unrolling the loop *k* times); and the value of some distinguished variable. These criteria are guided by syntactic hints inserted into a program prior to analysis; the analysis itself can choose to ignore these hints as a form of widening operator.

The analysis described by Mauborgne and Rival requires that the program is non-recursive; it fully inlines all procedure calls to attain context-sensitivity. Because the semantics they formulate does not contain an explicit representation of continuations, there is no way in their described system to achieve other forms of context-sensitivity (e.g., *k*-CFA) without heavily modifying their framework (cf. our discussion in Section 3.1). Our framework can express all of the sensitivities described by Mauborgne and Rival.

***k*-CFA.** There are several papers that describe various abstract interpretation-based approaches to *k*-CFA, including Ashley and Dybvig [1], Van Horn and Might [28], and Midtgaard and Jensen [17, 18]. Ashley and Dybvig [1] give an first-order collecting semantics formulation of *k*-CFA for a core Scheme-like language; they instrument both the concrete and abstract semantics with a *cache* that collects CFA information. The analysis as described in the paper is

[6] By which they mean that the abstract semantics say nothing about merging information, but the implementation does so anyway.

intractable (i.e., although it yields the same precision as *k*-CFA, the number of states remains exponential in the size of the program). They implement a more efficient, tractable version of the analysis independently from the formally-derived version, rather than deriving the tractable version directly from the formal semantics.

Van Horn and Might [28] also give a first-order collecting semantics formulation of *k*-CFA, in their case for the lambda calculus. An important contribution of this paper is a technique to abstract the infinite domains used for environments and semantic continuations using store-allocation (this is an alternative we could have used for our example analysis in Section 4 instead of CPS form). As with Ashley and Dybvig, the analysis as described in their paper does not yield a tractable analysis. Van Horn and Might describe a tractable version of their analysis (not formally derived from the language semantics) that uses a single, global store to improve efficiency. Van Horn and Might later extend their framework to the JavaScript language [10], though it remains intractable.

Midtgaard and Jensen [17] derive a tractable, demand-driven 0-CFA analysis for a core Scheme-like language using abstract interpretation. Their technique specifically targets 0-CFA, rather than general *k*-CFA. They employ a series of abstractions via Galois connections, the composition of which leads to the final 0-CFA analysis. In a later paper [18], Midtgaard and Jensen use abstract interpretation to derive another 0-CFA analysis to compute both call *and* return information. Rather than directly implementing the derived analysis they describe a technique to generate a set of constraints, which can then be solved using standard constraint-solving techniques.

7. Discussion and Conclusion

We have demonstrated the utility and practicality of our framework; there are several interesting directions to pursue from this point, such as:

Efficiency. Our JavaScript abstract interpreter shows that our framework enables tractable analysis for real-world languages, but the analysis performance still leaves room for improvement. Some of the performance hit comes from the inherent difficulty of analyzing a dynamic language like JavaScript; another portion comes from the flexibility of our framework—we cannot specialize the abstract interpreter to a specific form of control-flow sensitivity without compromising that flexibility. However, our current implementation has not been heavily optimized and there is still a lot of engineering work that can be done to improve performance (e.g., our implementation currently uses naive data structures for representing abstract domains and a naive iteration strategy for computing a fixed-point). More interesting is the possibility of incorporating formally well-founded algorithmic improvements such as *sparse analysis* [22]; recent work has shown that abstract interpretation can employ sparseness to increase analysis efficiency [21], but that work is confined to C-like languages and uses other assumptions that don't apply to our framework. We are investigating methods to incorporate sparse analysis into our JavaScript abstract interpreter and into our general framework.

Exploring Sensitivities. Our tunable framework makes it easy to explore a large space of control-flow sensitivities; this enables empirical comparisons of many existing forms of sensitivity to determine their relative effectiveness. It also enables exploration of new forms of control-flow sensitivity that potentially out-perform existing forms. For example, object-sensitivity (what we call allocation-site sensitivity in Section 4) was initially developed for statically-typed languages such as Java, where an object's type is fixed at allocation time. In JavaScript, an object is often modified dynamically with new fields and methods, changing its "type" over the course of the execution. Using an abstraction of an object's dynamic field insertions as the abstract trace could yield better results than object-sensitivity for dynamic languages. We are currently exploring this and similar ideas using our framework.

In summary, we have shown that control-flow sensitivity can be incorporated into an abstract interpreter as a widening operator. The abstract interpreter can be parameterized by the specific form of control-flow sensitivity desired; this means that the sensitivity can be modified independently from the rest of the analysis. These contributions provide a principled approach to the design and implementation of abstract interpretation-based control-flow sensitive analyses.

References

[1] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *TOPLAS*, 20(4), July 1998.

[2] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, 1979.

[3] A. Cortesi. Widening operators for abstract interpretation. In *Software Engineering and Formal Methods*, 2008.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *ESOP*, 2005.

[8] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[9] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010.

[10] D. V. Horn and M. Might. An Analytic Framework for JavaScript. *CoRR*, abs/1109.4467, 2011.

[11] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of logic in computer science*, volume 4. Oxford University Press, 1995.

[12] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7, 1977.

[13] G. A. Kildall. A unified approach to global program optimization. In *POPL*, 1973.

[14] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI*, June 2007.

[15] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28(2), Dec. 1990.

[16] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.

[17] J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *SAS*, 2008.

[18] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. *Information and Computation*, 211(0), 2012.

[19] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1), Jan. 2005.

[20] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), Mar. 2006.

[21] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, 2012.

[22] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *POPL*, 1977.

[23] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM annual conference*, 1972.

[24] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *TOPLAS*, 29(5), Aug. 2007.

[25] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[26] O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[28] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, 2010.