Widening for Control-Flow

Ben Hardekopf[1], Ben Wiedermann[2], Berkeley Churchill[3], and Vineeth Kashyap[1]

[1] University of California, Santa Barbara — {benh, vineeth}@cs.ucsb.edu
[2] Harvey Mudd College — benw@cs.hmc.edu
[3] Stanford University — bchurchill@cs.stanford.edu

Abstract. We present a parameterized widening operator that determines the control-flow sensitivity of an analysis, i.e., its flow-sensitivity, context-sensitivity, and path-sensitivity. By instantiating the operator's parameter in different ways, the analysis can be tuned to arbitrary sensitivities without changing the abstract semantics of the analysis itself. Similarly, the analysis can be implemented so that its sensitivity can be tuned without changing the analysis implementation. Thus, the sensitivity is an independent concern, allowing the analysis designer to design and implement the analysis without worrying about its sensitivity and then easily experiment with different sensitivities after the fact. Additionally, we show that the space of control-flow sensitivities induced by this widening operator forms a lattice. The lattice meet and join operators are the product and sum of sensitivities, respectively. They can be used to automatically create new sensitivities from existing ones without manual effort. The sum operation in particular is a novel construction, which creates a new sensitivity less precise than either of its operands but containing elements of both.

1 Introduction

A program analysis designer must balance three opposing characteristics: soundness, precision, and tractability. An important dimension of this tradeoff is *control-flow sensitivity*: how precisely the analysis adheres to realizable program execution paths. Examples from within this space include various types of *path sensitivity* (e.g., property simulation [11] and predicate abstraction [3]), *flow sensitivity* (e.g., flow-insensitive [4] and flow-sensitive [15]), and *context sensitivity* (e.g., k-CFA [27] and object sensitivity [21]). By tracking realizable execution paths more precisely, the analysis may compute more precise results but also may become less tractable. Thus, choosing the right control-flow sensitivity for a particular analysis is crucial for finding the sweet-spot that combines useful results with tractable performance.

We present a set of insights and formalisms that allow control-flow sensitivity to be treated as an independent concern, separately from the rest of the analysis design and implementation. This separation of concerns allows the analysis designer to empirically experiment with many different analysis sensitivities in a guaranteed sound manner, without modifying the analysis design or implementation. These sensitivities are not restricted to currently known strategies; the designer can easily develop and experiment with new sensitivities as well. Besides allowing manual exploration of potential new sensitivities, we also describe a mechanism to automatically create new sensitivities, based on the insight that the space of control-flow sensitivities forms a lattice. The

meet and join operators of this lattice can be used to construct novel sensitivities from existing ones without requiring manual intervention.

Key Insights. Our key insight is that control-flow sensitivity *is* a form of widening, and that we can exploit this to separate control-flow sensitivity from the rest of the analysis. This paper describes control-flow sensitivity as a widening operator parameterized by an equivalence relation that partitions states according to an abstraction of the program's history of computation. This widening-based view of control-flow sensitivity has both theoretical and practical implications: it generalizes and modularizes existing insights into control-flow sensitivity, and provides the analysis designer with a method for implementing and evaluating many possible sensitivities in a modular way.

Our work builds off of previous efforts to formalize control-flow sensitivity. A known technique to formalize a given form of control-flow sensitivity abstracts a program's concrete control flow as a specific abstract trace (i.e., some notion of the history of computation that led to a particular program point). There are many ways to design such an abstraction, including ad-hoc values that represent control-flow (e.g., the timestamps of van Horn and Might [29]), designed abstractions with a direct connection to the concrete semantics (e.g., the mementoes of Nielson and Nielson [22]), and calculated abstractions that result from the composition of Galois connections (e.g., the 0-CFA analysis derived by Midtgaard and Jensen [20]). Existing formalisms of control-flow abstraction are also tied to the notion of abstraction by partitioning [10]: the control-flow abstraction partitions the set of states into equivalence relations, the abstract values of which are merged.

Our formalisms follow this general approach (tracing and partitioning). However, prior work starts from a subset of known control-flow approximations (e.g, context-sensitivity [16, 22, 28], 0-CFA [20], or various forms of k-limiting and store value-based approximations [17, 24]) and seeks to formalize and prove sound those specific control-flow approximations for a given analysis. In addition, most prior work uses a calculation approach through a series of Galois connections that lead to a specific (family of) control-flow sensitivity. In contrast, our work provides a more general view that specifies a superset of the control-flow sensitivities specified by prior work and exposes the possibility of many new control-flow sensitivities, while simplifying the required formalisms and enabling a practical implementation based directly on our formalisms.

Contributions. This paper makes the following specific contributions:

- A new formulation of control-flow sensitivity as a widening operator, which generalizes and modularizes existing formulations based on abstraction by partitioning. This formulation leads to a method for designing and implementing a program analysis so that control-flow sensitivity is a separate and independent component. The paper describes several requirements on the form a semantics should take to enable separable control-flow sensitivity. Individually these observations are not novel; in fact, they may be well-known to the community. When collectively combined, however, they form an analysis design that permits sound, tunable control-flow approximation via widening. (Section 2)

– A novel way to automatically derive new control-flow sensitivities by combining existing ones. Our results follow from category theoretic constructions and reveal that the space of control-flow sensitivities forms a lattice. (Section 3)

– An in-depth example that applies our method to a language with mutable state and higher-order functions, creating a tractable abstract interpreter with separate and tunable control-flow sensitivity. We describe several example trace abstractions that induce well-known sensitivities. We also illustrate our example with an accompanying implementation, available in the supplemental materials.[4] (Section 4)

2 Separating Control-Flow Sensitivity from an Analysis

In this section we describe how to use widening to separate control-flow sensitivity from the rest of the analysis and make it an independent concern. We first establish our starting point: an abstract semantics that defines an analysis with no notion of sensitivity. We then describe a parameterized widening operator for the analysis and show how different instantiations of the parameter yield different control-flow sensitivities. Finally, we discuss some requirements on the form of semantics used by the analysis that make it amenable to describing control-flow sensitivity. The discussion in this section leaves the exact language and semantics being analyzed unspecified; Section 4 provides a detailed concrete example of these concepts for a specific language and semantics.

2.1 Starting Point

This subsection provides background and context on program analysis, giving us a starting point for our design. Nothing in this particular subsection is novel, the material is adapted from existing work [8]. For concreteness, we assume that the abstract semantics is described as a state transition system, e.g., a small-step abstract machine semantics; Section 2.4 will discuss more general requirements on the form of the semantics. The abstract semantics is formally described as a set of states $\hat{\varsigma} \in \Sigma^\sharp$ and a transition relation between states $\mathcal{F}^\sharp \subseteq \Sigma^\sharp \times \Sigma^\sharp$. The semantics uses a transition relation instead of a function to account for nondeterminism in the analysis due to uncertain control-flow (e.g., when a conditional guard's truth value is indeterminate, and so the analysis must take both branches). The set of states forms a lattice $\mathcal{L}^\sharp = (\Sigma^\sharp, \sqsubseteq, \sqcap, \sqcup)$. We leave the definition of states and the transition relation unspecified, but we assume that any abstract domains used in the states are equipped with a widening operator.[5]

The program analysis is defined as the set of all reachable states starting from some set of initial states and iteratively applying the transition relation. This definition is formalized as a least fixpoint computation. Let $\mathring{\mathcal{F}}^\sharp(S) \stackrel{\text{def}}{=} S \cup \mathcal{F}^\sharp(S)$, i.e., a relation that is lifted to remember every state visited by the transition relation $\mathcal{F}^\sharp$. The analysis of a program P is defined as $[\![P]\!]^\sharp \stackrel{\text{def}}{=} \mathbf{lfp}_{\Sigma_I^\sharp} \mathring{\mathcal{F}}^\sharp$, i.e., the least fixpoint of $\mathring{\mathcal{F}}^\sharp$ starting from an initial set of states $\Sigma_I^\sharp$ derived from P.

[4] `http://www.cs.ucsb.edu/~pllab/vmcai-14.zip`.

[5] If the domain is a noetherian lattice then the lattice join operator is a widening operator.

The analysis $[\![P]\!]^\sharp$ is intractable, because the set of reachable states is either infinite or, at the least, exponential in the number of nondeterministic transitions made during the fixpoint computation. The issue is control-flow—specifically, the nondeterministic choices that must be made by the analysis: which branch of a conditional should be taken, whether a loop should be entered or exited, which (indirect) function should be called, etc. The analysis designer at this point must either (1) bake into the abstract semantics a specific strategy for dealing with control-flow; or (2) ignore the issue in the formalized analysis design and use an ad-hoc strategy in the analysis implementation.

Our proposed widening operator is a means to formalize control-flow sensitivity in a manner that guarantees soundness, but does not require that any specific strategy must be baked into the semantics. On a practical level, it also allows the analysis designer to experiment with many different sensitivities without modifying the analysis implementation.

2.2 Widening Operator

Our goal is to *limit* the number of states contained in the fixpoint, while still retaining soundness. We do so by defining a widening operator for the fixpoint computation, which acts on entire sets of states rather than on individual abstract domains inside the states. This widening operator: (1) partitions the current set of reachable states into disjoint sets; (2) merges all of the states in each partition into a single state that over-approximates that partition; and (3) unions the resulting states together into a new set that contains only a single state per partition. The widening operator controls the performance and precision of the analysis by setting a bound on the number of states allowed: there can be at most one state per partition. Decreasing the number of partitions can speed up the fixpoint computation, thus helping performance, but can also merge more states together in each partition, thus hindering precision.

Formally, the widening operator for control-flow sensitivity is parameterized by a (unspecified) equivalence relation $\sim$ on abstract states. Given a widening operator $\triangledown$ on individual abstract domains, our new widening operator $\triangledown^\sharp$ is defined as:[6]

$$\triangledown^\sharp \in \mathcal{P}(\Sigma^\sharp) \times \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$$

$$A \mathbin{\triangledown^\sharp} B = \left\{ \bigvee_{\hat{\varsigma} \in X} \hat{\varsigma} \;\middle|\; X \in (A \cup B)/\sim \right\}$$

where for a set S the notation $S/\sim$ means the set of partitions of S according to equivalence relation $\sim$, and the widening operator $\triangledown$ on individual abstract domains is used to merge the states in each resulting partition into a single state. Note that if the number of partitions induced by $\sim$ is finite, then the number of states in each partition is also finite because we apply the widening operator at each step of the fixpoint computation.

Theorem 1 (WIDENING). *If the number of partitions induced by $\sim$ is finite, then $\triangledown^\sharp$ is a widening operator.*

[6] This widening operator is reminiscent of an existing notion called *disjunctive abstract domains* [6]. We discuss the relation between the two further in Section 5.

Proof. Follows from the definition of a widening operator [9].

We now lift the transition relation $\mathcal{F}^\sharp$ in a similar fashion as before, except instead of using set union we use our widening operator: $\overset{\triangledown}{\mathcal{F}}^\sharp(S) \overset{\text{def}}{=} S \;\triangledown^\sharp\; \mathcal{F}^\sharp(S)$. Then the control-flow sensitive abstract semantics is defined as $[\![P]\!]^\sharp_\triangledown \overset{\text{def}}{=} \mathbf{lfp}_{\Sigma^\sharp_I} \overset{\triangledown}{\mathcal{F}}^\sharp$.

Even though we have not specified the equivalence relation that parameterizes the widening operator, we can still prove the soundness of the analysis. Informally, because the widening operator merges the states within each partition using $\triangledown$, the reachable states using $\overset{\triangledown}{\mathcal{F}}^\sharp$ over-approximate the reachable states using $\overset{\circ}{\mathcal{F}}^\sharp$. Thus, the control-flow sensitive abstract semantics is sound with respect to the original abstract semantics:

Theorem 2 (SOUNDNESS).

$$\gamma([\![P]\!]^\sharp) \;\subseteq\; \gamma([\![P]\!]^\sharp_\triangledown)$$

Proof. We must show that (1) the least fixpoint denoted by $[\![P]\!]^\sharp_\triangledown$ exists; and (2) it over-approximates $[\![P]\!]^\sharp$.

1. The existence of the fixpoint follows from part 2 of the definition of a widening operator as given by Cousot and Cousot [9, def. 9.1.3.3].
2. That the widened fixpoint over-approximates the original fixpoint follows from part 1 of the definition of a widening operator as given by Cousot and Cousot [9, defs. 9.1.3.1–9.1.3.2].

2.3 Control-Flow Sensitivity

It remains to show how our widening operator determines the control-flow sensitivity of the analysis. The determining factor is how the states are partitioned, which is controlled by the specific equivalence relation on states $\sim$ that parameterizes the widening operator. The question is, what constitutes a good choice for the equivalence relation? For Theorem 1 to hold, it must induce a finite number of partitions, but what other characteristics should it have? Our goal is tractability with a minimal loss of precision; this means we should try to partition the states so that there are a tractable number of partitions *and* the states within each partition are as similar to each other as possible (to minimize the information lost to merging).

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 heuristic of similarity is exactly the one used by existing control-flow sensitivities, such as flow-sensitive maximal fixpoint, k-CFA, object-sensitivity, property simulation, etc. These sensitivities 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 and merge the states during the analysis.

Therefore, the widening operator should partition the set of states according to their control-flow sensitivity approximation:

$$\hat{\varsigma}_1 \sim \hat{\varsigma}_2 \iff \pi_{\hat{\tau}}(\hat{\varsigma}_2) = \pi_{\hat{\tau}}(\hat{\varsigma}_2)$$

where each state contains an *abstract trace* $\hat{\tau}$ describing some abstraction of the execution history, and $\pi_{\hat{\tau}}(\hat{\varsigma})$ projects a state's abstract trace. This definition causes the widening operator to merge all states with the same trace, i.e., all states with the same approximate execution history. The widened analysis can be defined without specifying a particular abstract trace domain; different trace domains can be plugged in after the fact to yield different sensitivities.

Trace Abstractions. We have posited that control-flow sensitivity is based on an abstraction of the execution history of a program, called a trace. This implies that the trace abstraction is related to the *trace-based* concrete collecting semantics, which contains all reachable execution paths, i.e., sequences of states, rather than just all reachable states. An abstract trace is an abstraction of a set of paths in the concrete collecting semantics. For example, a flow-sensitive trace abstraction records the current program point, abstracting all paths that reach that program point. A context-sensitive trace abstraction additionally records the invocation context of the current function, abstracting all paths that end in that particular invocation context (e.g., as in Nielson and Nielson's mementoes [22]). Different forms of context-sensitivity define the abstract "context" differently: for example, traditional k-CFA defines it as the last k call-sites encountered in the concrete 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.

We note that it is not necessary for the trace abstraction to soundly approximate the concrete semantics for the resulting analysis to be sound. The trace abstraction is a heuristic for partitioning the states; as long as the number of elements in the trace abstraction domain is finite (and hence the number of partitions enforced by the widening operator is finite), the analysis will terminate with a sound solution. In fact, it isn't strictly necessary for $\sim$ to be based on control-flow at all—exploring other heuristics for partitioning states would be in interesting avenue for future work.

2.4 Semantic Requirements

To benefit from widening-based control-flow sensitivity, an abstract semantics must satisfy certain requirements. To abstract control, the analysis must be able to introduce new program execution paths that over-approximate existing execution paths. To make this possible, we argue that there should be some explicit notion in the program semantics of the "rest of the computation"—i.e., a continuation. When the analysis abstracts control, it is abstracting these continuations. The explicit control-flow representation can take a number of possible forms. For example, it could 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, any two distinct states must have a join, and (according to our requirement) this joined state must contain a continuation that over-approximates the input states' continuations. Thus, by joining states the analysis approximates control as well as data.

Some forms of semantics do not meet this requirement, including various forms proposed as being good foundations for abstract interpretation [18, 25, 26]. 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. This means that there is no way to abstract and over-approximate control-flow; the analysis must use whatever control-flow the original semantics specifies (or, alternatively, use ad-hoc strategies baked into the analysis implementation to silently handle control-flow sensitivity). Some limited forms of control-flow sensitivity may still be expressed when the analysis takes care to join only those states that already have the same continuation (e.g., flow-sensitive maximal fixpoint), but many other forms (e.g., k-CFA or other forms of context-[in]sensitivity) remain difficult to express.

Continuations vs. Control-Flow Graphs. It is common in dataflow analysis to use a control-flow graph (CFG) to represent a program's control-flow rather than embedding continuations inside states. CFG edges can be seen as *externalized* continuations, i.e., continuations removed from the states and reified as a separate data structure [1]. Our method will work with CFGs, but with some restrictions and caveats due to the fact that states are divorced from their continuations, and thus joining states does not join continuations. The basic problem is that it can be difficult to associate the right continuation with the right state. Sometimes the analysis needs to know under what context it entered a region of code, to compute the next transition (for example, which caller to return to for an indirect call). When continuations are embedded inside states, this information is obvious; when continuations are separate, it becomes messy and ad-hoc. The problem of associating states and continuations usually manifests for interprocedural edges; an interesting hybrid scheme would be to use CFGs intraprocedurally, but internalized continuations for interprocedural control-flow.

3 Relating and Combining Sensitivities

One of the goals of this work is to make it easier for analysis designers to experiment with new trace abstractions. To this end, it would be useful to systematically create new trace abstractions from existing ones, and to understand how trace abstractions relate to one another.

An obvious way to combine trace abstractions in order to create a new form of control-flow sensitivity is to take their *product*.[7] Given two trace abstractions, one constructs their product by taking the cartesian product of the corresponding sets and defining the update function to act pairwise on the resulting tuple. A less obvious method of combining trace abstractions is to take their *sum*. This is a novel way to create new control-flow sensitivities that has not been presented before. Think of a trace abstraction as allowing the analysis to decide whether two abstract states computed during the analysis should be joined. Informally, the product of two trace abstractions joins two states only if *both* traces agree that the states should be joined, whereas the sum of two trace abstractions joins two states if *either* trace determines that the states should be joined.

[7] Another interesting combination to explore would be the *reduced product*, however it is not in general possible to automatically derive the reduced product of two domains [9, §10].

In the next section we describe how to construct the sum of two traces. We then show that sum and product are the join and meet operations of a lattice of control-flow sensitivities. This construction suggests new control-flow sensitivities that could be used in practice and also enables a fully automated exploration of control-flow sensitivities that complements manual exploration. The supplementary material contains an implementation of the product and sum operators described here, as part of the implementation of the example abstract semantics described in Section 4.

3.1 Sums of Trace Abstractions

While the product of two traces is obvious, constructing the sum is unintuitive. We formally define a trace abstraction as a (unspecified) finite set $\Theta^\sharp$, a distinct element $\mathbf{1} \in \Theta^\sharp$ that acts as an initial trace for the analysis, and a trace update function $\tau_{update} : (\Sigma^\sharp \times \Theta^\sharp) \uplus \mathbf{1} \to \Theta^\sharp$ that specifies how the trace changes at each statement transition in the abstract semantics. The pair $(\Theta^\sharp, \tau_{update})$ is the object we call a trace abstraction. When discussing multiple trace abstractions we use Θ_X, τ_X, and $\mathbf{1}_X$ to denote the $\Theta^\sharp$, τ_{update}, and initial trace for each trace abstraction X.

A naive attempt to construct the sum would use the disjoint union of the trace abstractions' underlying sets. However, this attempt fails because each trace abstraction has a unique initial trace, and thus the disjoint union does not constitute a valid trace abstraction. To create a valid sum $X + Y$ from trace abstractions X and Y, we must create a new set Θ_{X+Y} whose initial trace $\mathbf{1}_{X+Y} \in \Theta_{X+Y}$ "agrees" with both $\mathbf{1}_X$ and $\mathbf{1}_Y$, in a sense that we formalize below. The sum transition function τ_{X+Y} must also "agree" with both τ_X and τ_Y, in the same sense. The central insight behind the sum construction is to construct an equivalence relation between elements of Θ_X and Θ_Y, and let the elements of Θ_{X+Y} be the corresponding equivalence classes. Then $\mathbf{1}_{X+Y}$ and τ_{X+Y} agree with the individual trace abstractions X and Y if they produce equivalence classes that contain the same elements that would have been produced by X and Y individually. It remains to describe how the analysis creates these equivalence classes: they cannot be constructed before the analysis begins, rather the analysis constructs them dynamically (i.e., as it executes) in the following way.

Definition 1. *Let X and Y be trace abstractions and $\hat{\varsigma}$ be an abstract state. Inductively define an equivalence relation $\sim$ on the disjoint union $\Theta_X \uplus \Theta_Y$ by taking the symmetric, reflexive and transitive closure while applying the following rules:*

$$\mathbf{1}_X \sim \mathbf{1}_Y$$

$$\frac{i, j \in \{X, Y\} \qquad a \in \Theta_i \qquad b \in \Theta_j \qquad a \sim b}{\tau_i(\hat{\varsigma}, a) \sim \tau_j(\hat{\varsigma}, b)}$$

The sum $X + Y$ has underlying set $\Theta_{X+Y} = (\Theta_X \uplus \Theta_Y)/\sim$, i.e., the set of equivalence classes of $\Theta_X \uplus \Theta_Y$ according to $\sim$. The equivalence relation is defined by construction so that τ_X and τ_Y will always agree on which equivalence class of $\hat{\tau}_X \uplus \hat{\tau}_Y$ to transition to. We now define the disjoint union transition function τ_{X+Y}. For $\hat{\tau} \in \Theta_X$, let $[\hat{\tau}]$ denote the equivalence class of $\hat{\tau}$ in Θ_{X+Y}. The function τ_{X+Y} is defined as follows: for any $\hat{\varsigma} \in \Sigma^\sharp$ and $[\hat{\tau}] \in \Theta_{X+Y}$, pick some $\hat{\tau}' \in \Theta_X$ so that $\hat{\tau}' \in [\hat{\tau}]$. Then $\tau_{X+Y}(\hat{\varsigma}, [\hat{\tau}]) = [\tau_X(\hat{\varsigma}, \hat{\tau}')]$. The

equivalence relation ensures this is well defined for any valid choice of $\hat{\tau}'$. By symmetry, the same applies to $\hat{\tau} \in \Theta_Y$.

The essence of this definition is that it causes the following diagram to commute, making τ_X and τ_Y agree with τ_{X+Y}:

$$\begin{array}{ccccc}
\Sigma^\sharp \times \Theta_X \uplus \mathbf{1} & \xrightarrow{\mathbf{1}\mapsto\mathbf{1}} & \Sigma^\sharp \times \Theta_{X+Y} \uplus \mathbf{1} & \xleftarrow{\mathbf{1}\leftarrow\mathbf{1}} & \Sigma^\sharp \times \Theta_Y \uplus \mathbf{1} \\
\big\downarrow{\scriptstyle \tau_X} & & \big\downarrow{\scriptstyle \tau_{X+Y}} & & \big\downarrow{\scriptstyle \tau_Y} \\
\Theta_X & \xrightarrow[[\cdot]]{} & \Theta_{X+Y} & \xleftarrow[[\cdot]]{} & \Theta_Y
\end{array}$$

$(\hat{\varsigma},x) \mapsto (\hat{\varsigma},[x])$ $\quad$ $(\hat{\varsigma},[y]) \leftarrow (\hat{\varsigma},y)$

$X + Y$ is a trace abstraction which is less precise than both X and Y individually, but still contains some information from both. When implemented, the definition of the equivalence relation is unknown until runtime, where it is incrementally discovered by the analysis. Initially, the relation is the one forcing $\mathbf{1}_X \sim \mathbf{1}_Y$. At each iteration of the fixpoint calculation the functions τ_X and τ_Y are computed, and the results are used to discover more equivalences.

From another perspective, one can also view a trace abstraction X as a finite automaton with a set of automaton states Θ_X, a transition function τ_X, and an initial point $\mathbf{1}_X$. The input alphabet is the set $\Sigma^\sharp$. We were surprised to discover that, from this perspective, our definition of summing trace abstractions corresponds exactly to a widening operator on finite automata described by Bartzis and Bultan [5]. Their operator was designed to provide a "less precise" finite automaton that accepts a larger language than both of its inputs.

3.2 Trace Abstractions Form a Lattice

The sum and product operations described above are dual to each other in a special way: they are the join and meet operations of a lattice. The lattice partial order is based on the precision of the trace abstractions. Using the notation from the previous section, we say that a trace abstraction X is *more precise* than Y (written as $X \leq Y$) if there exists a relation on the corresponding automata satisfying certain properties.

Definition 2. *$X \leq Y$ if there is a relation $R \subset Y \times X$ such that*

1. $(\mathbf{1}_Y, \mathbf{1}_X) \in R$
2. $(y, x) \in R$ *implies for all* $\hat{\varsigma} \in \Sigma^\sharp$, $(\tau_Y(\hat{\varsigma}, y), \tau_X(\hat{\varsigma}, x)) \in R$
3. *R is injective, meaning* $(y, x) \in R$ *and* $(y', x) \in R$ *implies* $y = y'$.

The relation R forces Y and X to behave in the same way, but also requires that X has "more" members than Y. It can be likened to an injective function. The relation $\leq$ is a preorder; by implicitly taking equivalence classes it becomes a partial order. Intuitively, $X < Y$ corresponds to the intuition "X is strictly more precise than Y", in every way of measuring it. Members of the same equivalence class correspond to families of trace abstractions that provide exactly the same precision.

Theorem 3. *The space of trace abstractions form a lattice, where sum corresponds to join and product corresponds to meet.*

Proof. The proof follows from elementary results in category theory, order theory and our definitions. The details are given in the supplementary materials.

Use of Category Theory. Category theory provides useful constructions that apply in general settings. We arrived at our construction of the sum operator and the lattice of sensitivities via category theory, because they were non-obvious without this perspective. We used category theory to derive the definition for sums of trace abstractions and prove the theorems elegantly, and we suspect it can be used to achieve further insights into combining sensitivities. In our supplementary material we detail how we used it to arrive at our results.

4 Analysis Design Example

In this section we give a detailed example of how to build an abstract interpreter with separate and tunable control-flow sensitivity. Our example picks up in the middle of the design process: an analysis designer has formally defined an abstract semantics that is amenable to defining control-flow sensitivity (cf. Section 2) and the semantics has been proven sound with respect to a concrete semantics.[8] We show how to extend this analysis to support tunable control flow and how to easily and modularly tune the control-flow sensitivity of the resulting analysis. The supplementary material contains an implementation of this example analysis written in Scala.

4.1 Syntax

Figure 1 gives the syntax of a small but featureful language. It contains integers, higher-order functions, conditionals, and mutable state. As discussed in Section 2.4, tunable control-flow sensitivity requires an explicit representation of a program's control. For this example we chose to make control-flow explicit in the program syntax, hence we use continuation-passing style (CPS). We assume that the CPS syntax is the result of a CPS-translation from a programmer-facing, direct-style syntax.

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

$$L \in \mathit{Lam} ::= \lambda \overrightarrow{x} \,.\, S$$

$$T \in \mathit{Trivial} ::= [n_1..n_2] \mid x \mid L \mid T_l \oplus T_r$$

$$S \in \mathit{Serious} ::= \textbf{let}\ x = T\ \textbf{in}\ S \mid \textbf{set}\ x = T\ \textbf{in}\ S \mid \textbf{if}\ T\ S_t\ S_f \mid x(\overrightarrow{T})$$

Fig. 1: Continuation-passing style (CPS) syntax for the example language. Vector notation denotes an ordered sequence. The notation $[n_1..n_2]$ denotes nondeterministic choice from a range of integers, to simulate, e.g., user input.

As usual for CPS [23], expressions are separated into two categories: *Trivial* and *Serious*. *Trivial* expressions are guaranteed to terminate and to have no side-effects; *Serious* expressions make no such guarantees. Functions take an arbitrary number of

[8] For reasons of space we omit the concrete semantics and soundness proof; neither are novel.

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 (including a **halt** continuation that terminates evaluation). We assume that it is possible to syntactically disambiguate among calls 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 Original Abstract Semantics

The original abstract semantics defines a computable approximation of a program's behavior using a small-step abstract machine. Figure 2a describes the semantic domains (for now, ignore the boxed elements[9]). An abstract state consists of a set of *Serious* expressions $\overline{S}$ (which represents the set of expressions that might execute at the current step), an abstract environment $\hat{\rho}$, and an abstract store $\hat{\sigma}$. Because the language is dynamically typed, any variable may be an integer or a closure at any time. Thus, abstract values are a product of two abstractions: one for integers and one for closures.[10] Integers are abstracted with the constant propagation lattice $\mathbb{Z}^\sharp = \mathbb{Z} \cup \{\top_{\mathbb{Z}^\sharp}, \bot_{\mathbb{Z}^\sharp}\}$, and closures are abstracted with the powerset lattice of abstract closures. The analysis employs a finite address domain $\mathit{Address}^\sharp$ and a function *alloc* to generate abstract addresses; we leave these elements unspecified for brevity.

Figure 2b describes the semantic function $\hat{\eta}$, which abstractly interprets *Trivial* expressions. Literals evaluate to their abstract counterparts, injected into a tuple. Variable lookup joins all the abstract values associated with that variable. The abstract transition function $\mathcal{F}^\sharp$ transforms abstract states. 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 semantics is nondeterministic, meaning that one state may potentially transition to multiple states. The semantics also employs *weak updates*: 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 always uses weak updates.

Figure 2c describes the abstract semantics of *Serious* expressions. The unboxed features are standard. Note the sources of non-determinism: An **if** statement may lead to multiple states (i.e., when the condition's abstract value is not precise enough to send the abstract interpretation down only one brach). A function call also may lead to multiple states (i.e., when evaluating the function's name leads to a set of closures, each of which is traced by the abstract interpretation).

The full analysis is defined as the reachable states abstract collecting semantics of *Serious* evaluation: $[\![S]\!]^\sharp = \mathbf{lfp}^{\circ}_{\Sigma^\sharp_{\mathcal{I}}} \mathcal{F}^\sharp$. This analysis is sound and computable, however it

[9] All the unboxed elements define the original abstract semantics. The boxed elements describe the extensions that support parameterized control-flow sensitivity; they are described in Section 4.3.

[10] For brevity, we omit error-handling semantics. We also sometimes omit one part of the tuple, when the meaning is clear from the context (e.g., when interpreting *Serious* values in Figure 2c).

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

$$\begin{aligned}
\hat{\varsigma} \in \Sigma^\sharp &= \mathcal{P}(\mathit{Serious}) \times \mathit{Env}^\sharp \times \mathit{Store}^\sharp \times \boxed{\Theta^\sharp} && \text{(abstract states)} \\
\hat{\rho} \in \mathit{Env}^\sharp &= \mathit{Variable} \to \mathcal{P}(\mathit{Address}^\sharp) && \text{(environments)} \\
\hat{\sigma} \in \mathit{Store}^\sharp &= \mathit{Address}^\sharp \to \mathit{Value}^\sharp && \text{(stores)} \\
\widehat{clo} \in \mathit{Closure}^\sharp &= \boxed{\Theta^\sharp} \times \mathit{Env}^\sharp \times \mathit{Lam} && \text{(closure values)} \\
\hat{v} \in \mathit{Value}^\sharp &= \mathbb{Z}^\sharp \times \mathcal{P}(\mathit{Closure}^\sharp) && \text{(abstract values)} \\
\mathcal{F}^\sharp \in \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp) &= \text{Figure 2c} && \text{(transition function)}
\end{aligned}$$

(a) Abstract semantic domains.

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

$$\begin{aligned}
\hat{\eta}([n_1..n_2], \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \alpha([n_1..n_2]), \emptyset \rangle \\
\hat{\eta}(x, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \bigsqcup_{\hat{a} \in \hat{\rho}(x)} \hat{\sigma}(\hat{a}) \\
\hat{\eta}(\lambda\overrightarrow{x} \,.\, S, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \langle \bot_{\mathbb{Z}^\sharp}, \{\langle \boxed{\hat{\tau}}, \hat{\rho}, \lambda\overrightarrow{x} \,.\, S \rangle\} \rangle \\
\hat{\eta}(T_l \oplus T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau}) &= \hat{\eta}(T_l, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \;\hat{\oplus}\; \hat{\eta}(T_r, \hat{\rho}, \hat{\sigma}, \hat{\tau})
\end{aligned}$$

(b) Abstract *Trivial* evaluation.

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

(c) Abstract transition function $\mathcal{F}^\sharp$, where $[\![\cdot]\!] = \hat{\eta}(\cdot, \hat{\rho}, \hat{\sigma}, \hat{\tau})$ and a fresh address $\hat{a}'$ 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 $\mathcal{F}^\sharp(\hat{\varsigma}) = \overline{\langle \{S'\}, \hat{\rho}', \hat{\sigma}', \hat{\tau}' \rangle}$.

Fig. 2: A standard abstract semantics over a simple abstract value domain (constant- and closure-propagation). The boxed elements indicate what extensions are necessary for this semantics to support parameterized control-flow sensitivity.

is still intractable because the set of reachable states grows exponentially with the number of nondeterministic branch points. To make this analysis tractable requires some form of control-flow sensitivity.

4.3 Tunable Control-Flow Sensitivity

We now extend the abstract semantics for our language to express tunable control-flow sensitivity. As described in Section 2, we extend the definition of abstract states to include a trace abstraction domain $\hat{\tau} \in \Theta^\sharp$. We leave the trace abstraction domain unspecified; the specific instantiation of the trace abstraction domain will determine the analysis control-flow sensitivity, as exemplified in Appendix A.

We make three changes to the abstract semantics of the previous section to integrate trace abstractions into the semantics; these are represented by the boxed elements in Figure 2. First, we add the trace abstraction domain to the abstract state definition. Next, we modify $\mathcal{F}^\sharp$ to operate on this new domain. This change gives the trace update mechanism access to all the data needed to compute a new abstract trace. Finally, we extend 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.

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

$$\begin{aligned} \hat{\tau} &\in \Theta^\sharp \\ \tau_{stmt} &\in \Sigma^\sharp \times Serious \to \Theta^\sharp \\ \tau_{call} &\in \Sigma^\sharp \times Closure^\sharp \to \Theta^\sharp \end{aligned}$$

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. The program analysis is defined as the widened reachable states abstract collecting semantics, $[\![S]\!]^\sharp_\nabla = \mathbf{lfp}_{\Sigma^\sharp_I} \overset{\nabla}{\mathcal{F}}{}^\sharp$, where the equivalence relation $\sim$ is the one given in Section 2.3.

The precision and performance of the analysis depend on the particular choice of trace abstraction for control-flow sensitivity. In this section, we present one illustrative example. Appendix A gives fives more. We could define many more, but our choices are sufficient to demonstrate the utility and flexibility of our method. Note that given these six definitions, we can automatically construct many distinct sensitivities using various combinations of the sum and product operators defined in Section 3.

Example: Flow-sensitive, stack-based k-CFA analysis In dataflow analysis, k-CFA is usually defined as having a stack- like behavior: 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.

Algorithm 1 Flow-sensitive, stack-based k-CFA

$$\hat{\tau} \in \Theta^\sharp = \mathit{Label} \times \mathit{Label}^\star$$

$$\tau_{stmt}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$

$$\tau_{call}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \hat{\tau}_c, \_, \lambda \mathbf{y} \,.\, S_c \rangle) = \langle S_c^\ell,\ \hat{\tau}' \rangle$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

5 Related Work

In abstract interpretation, there is a relatively small but dedicated body of research on trace abstraction and on formalizing control-flow sensitivity as partitioning. What distinguishes our work and prior efforts is a different focus: prior work focuses on the *integration* of control-flow abstractions into an existing analysis; our work focuses on the *separation* of control-flow abstractions from an existing analysis, so that it is easier for analysis designers to experiment with different control-flow sensitivities. In this section, we discuss the implications of these differences. Broadly, no prior work has couched control-flow sensitivity in terms of a widening operator based on abstractions of the program history, which permits a simpler, more general, and more tunable formulation of control-flow sensitivity.

Trace Partitioning. Our work is similar in some respects to the trace partitioning work by Mauborgne and Rival [17, 24], which itself builds on the abstraction-by-partitioning of control-flow by Handjieva and Tzolovski [13]. 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 reachable states collecting semantics, making it difficult to express control-flow sensitivity; and (2) designed to silently merge information at control-flow join points[11]—what in dataflow analysis is called "flow-sensitive maximal fixpoint" [14]. They propose a method to postpone these silent merges when doing so

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

can increase precision; effectively they add a controlled form of path-sensitivity. They formalize their technique as a series of Galois connections.

Mauborgne and Rival describe a denotational semantics-based analysis that can use three 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 and merge information anyway as a form of widening operator. This feature is a form of *dynamic partitioning*, where the choice of partition is made as the analysis executes. Our sum abstraction (Section 3.1) is another form of dynamic partitioning.

The analysis described by Mauborgne and Rival requires that the program is non-recursive; it fully inlines all procedure calls to attain complete 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, including 0-CFA, i.e., context-insensitive analysis) without heavily modifying their design, implementation, and formalisms (cf. our discussion in Section 2.4). Because our method seeks more generality, it can express all of the sensitivities described by Mauborgne and Rival.

Predicate Abstraction. Fischer et al. [12] propose a method to join dataflow analysis with predicate abstraction using *predicate lattices* to gain a form of tunable intra-procedural path-sensitivity. At a high level these predicate lattices perform a similar "partition and merge" strategy as our own method. However, our method is more general: we can specify many more forms of control-flow sensitivity due to our insights regarding explicit control state. One can consider their work as a specific instantiation of our method using predicates as the trace abstraction. On the other hand, Fisher et al. use predicate refinement to *automatically* determine the set of predicates to use, which is outside the current scope of our method. In order to do the same, our method would need to add a predicate refinement strategy.

Context sensitivity. There are several papers that describe various abstract interpretation-based approaches to specific forms of context sensitivity, including Nielson and Nielson [22], Ashley and Dybvig [2], Van Horn and Might [29], and Midtgaard and Jensen [19, 20]. Nielson and Nielson describe a form of context-sensitivity based on abstractions of the history of a program's calls and returns [22]. Although this formulation is separable, it is not as general as the one described in this paper. For example, it cannot capture calls and returns in obfuscated binaries (which may contain no explicit calls and returns); to capture such behavior, a different formulation similar to property simulation is required [16]. Our parameterized, widening-based approach we describe is general enough to capture *either* of these formulations (and many more).

Ashley and Dybvig [2] give a reachable states 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 intractable (i.e., although it yields the same precision as k-CFA, the number of states remains exponential in the size of the program). Ashley and Dybvig implement

a tractable, flow-insensitive 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 [29] also give a method for constructing analyses, using an abstract machine-based, reachable states collecting semantics of the lambda calculus. Their analysis includes a specification of k-CFA. An important contribution of their 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 directly 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, but disallows flow-sensitive analysis because it computes a single solution for the entire program.

Midtgaard and Jensen [19] 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 [20], Midtgaard and Jensen derive another 0-CFA analysis to compute both call *and* return information. Our example semantics of Section 4 bears a resemblance to Midtgaard and Jensen's (and to van Horn and Might's machine construction), but our goals differ. Our example illustrates how to achieve a sound analysis with *arbitrary* control-flow sensitivity, without having to derive the soundness for each sensitivity.

Relation to Disjunctive Domains The widening operator we define in Section 2 is reminiscent of an existing notion called *disjunctive abstract domains* [6]. This techniqe is used to augment the precision of a convex abstract domain by using the corresponding *powerset* domain, i.e., an abstract value is now a *set* of absract elements instead of a single element. The analysis interpret a set of elements as a disjunction. However, the powerset domain (also known as the disjunctive domain) is exponentially larger than the original domain, which makes the analysis intractable. Thus, we need to define a widening operator $\triangledown$ on the powerset domain that accelerates convergence by occasionally *merging* elements instead of unioning them.

The relation to our method is clear: our domain for the abstract collecting semantics is the powerset domain of abstract states (which can be interpreted as a disjunctive domain) and our widening operator accelerates convergence by occasionally merging states (using the state widening operator) instead of unioning states. The novelty of our method comes from two aspects. First, we link the notions of *widening* and *control-flow sensitivity*, which have never been related before. This link is possible when control is explicit in the abstract states as discussed in Section 2.4; thus joining states (our widening) necessarily abstracts control as well as data. It is exactly this behavior that enables control-flow sensitivity. Second, we show how to use this relation to create *tunable* control-flow sensitivity, as discussed in Section 2.3 and exemplified in Section 4. By making the relation between widening and control-flow sensitivity explicit and formal, we are able to take advantage of this new insight to create a practical method for tunable precision.

6 Conclusions and Future Work

We have presented a method for program analysis design and implementation that allows the analysis designer to parameterize over control-flow abstractions. This separation of concerns springs from a novel theoretical insight that control-flow sensitivity is induced by a widening operator parameterized by trace abstractions. Our method makes it easier for an analysis designer to specify, implement, and experiment with many forms of control-flow sensitivity, which is critical for developing new, practical analyses. For example, there are many popular hybrid object-oriented / functional languages (e.g., JavaScript, C#, Scala, etc.), but no one knows whether we should analyze them with control-flow sensitivities that were developed in a functional context (e.g., k-CFA) or an object-oriented context (e.g., object-sensitivity) or some hybrid thereof. Our method can be used to quickly experiment with a wide range of possibilities to gain insight into this question.

Our perspective on the space of trace abstractions as a category also enabled new insights into automatically constructing and combining trace abstractions in novel ways to achieve new forms of control-flow sensitivity. Our future work involves exploring these ideas further, for example, using combinatorial optimization to explore the vast space of possible trace abstractions. Also, as observed earlier, there is really nothing in our method specific to control-flow: our method applies to *any* property of a program that can be abstracted and that might be useful to partition the analysis state-space. We will also explore possible heuristics besides control-flow.

References

1. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1992)
2. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(4) (Jul 1998)
3. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2001)
4. Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (1979)
5. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Computer Aided Verification (CAV). pp. 321–333 (2004)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 269–282 (1979)
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyser. In: European Symposium on Programming (ESOP) (2005)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (1979)
10. Cousot, P., Cousot, R.: Invited Talk: Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In: IEEE Computer Society International Conference on Computer Languages (1994)

11. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2002)
12. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. In: European Software Engineering Conference (2005)
13. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Symposium on Static Analysis (SAS) (1998)
14. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7 (1977)
15. Kildall, G.A.: A unified approach to global program optimization. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (1973)
16. Lakhotia, A., Boccardo, D.R., Singh, A., Manacero, A.: Context-sensitive analysis of obfuscated x86 executables. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM) (2010)
17. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: European Symposium on Programming (ESOP) (2005)
18. Metayer, D.L., Schmidt, D.: Structural operational semantics as a basis for static program analysis. ACM Computing Surveys 28, 340–343 (1996)
19. Midtgaard, J., Jensen, T.: A calculational approach to control-flow analysis by abstract interpretation. In: Symposium on Static Analysis (SAS) (2008)
20. Midtgaard, J., Jensen, T.P.: Control-flow analysis of function calls and returns by abstract interpretation. Information and Computation 211(0), 49 – 76 (2012)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(1) (Jan 2005)
22. Nielson, F., Nielson, H.R.: Interprocedural control flow analysis. In: European Symposium on Programming (ESOP) (1999)
23. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM Annual Conference (1972)
24. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(5) (Aug 2007)
25. Schmidt, D.A.: Natural-Semantics-Based abstract interpretation. In: International Static Analysis Symposium (SAS) (1995)
26. Schmidt, D.A.: Abstract interpretation of small-step semantics. Lecture Notes in Computer Science 1192, 76–99 (1997)
27. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 1991), technical Report CMU-CS-91-145
28. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2011)
29. Van Horn, D., Might, M.: Abstracting abstract machines. In: ACM SIGPLAN International Conference on Functional programming (ICFP) (2010)

A Examples of Control-Flow Sensitivities

In this section, we define example control-flow sensitivities for the analysis given in Section 4.

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. Using our method, the analysis designer can specify flow-insensitive analysis by making the $\Theta^\sharp$ domain a single value, so that all states will necessarily have the same abstract trace and hence belong to the same singular partition.

Analysis 2 Flow-insensitive, context-insensitive

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= \mathbf{1} \\ \tau_{stmt}(\_, \_) &= \mathbf{1} \\ \tau_{call}(\_, \_) &= \mathbf{1} \end{aligned}$$

We use an underscore to indicate an arbitrary value in the function parameters. The widened analysis joins every state into one single global state, i.e., the fixpoint computation continually adds information to a single program-wide state until that state converges.

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 $\Theta^\sharp$ domain the set of program labels and updating the trace at each step to be the current program point.

Analysis 3 Flow-sensitive, context-insensitive

$$\begin{aligned} \hat{\tau} \in \Theta^\sharp &= \mathit{Label} \\ \tau_{stmt}(\_, S) &= S^\ell \\ \tau_{call}(\_, \langle \_, \_, \lambda \mathbf{y} \,.\, S_c \rangle) &= S_c^\ell \end{aligned}$$

The abstract semantics at each step collects all states at the same program point and joins them together, constraining the maximum number of abstract states to be the number of program points. In the dataflow analysis community this is called the flow-sensitive *maximal fixpoint analysis* (MFP).

FS + traditional *k*-CFA analysis. Traditional k-CFA [27] 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 translating **let** and **set** into function calls. The analysis designer can specify flow-sensitive k-CFA by making the $\Theta^\sharp$ 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 4 Flow-sensitive, k-CFA

$$\hat{\tau} \in \Theta^\sharp = \mathit{Label} \times \mathit{Label}^\star$$

$$\tau_{\mathit{stmt}}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell, \pi_2(\hat{\tau}) \rangle$$

$$\tau_{\mathit{call}}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \_, \_, \lambda \boldsymbol{y} \,.\, S_c \rangle) = \langle S_c^\ell, \hat{\tau}' \rangle$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (S^\ell :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

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.

FS + *k*-allocation-site sensitive analysis Object-sensitivity [21] 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 [28] 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 abstract trace.

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

$$\hat{\tau} \in \Theta^\sharp = \mathit{Label} \times \mathit{Address}^{\sharp\star}$$
$$\tau_{\mathit{stmt}}(\langle \_, \_, \_, \hat{\tau} \rangle, S) = \langle S^\ell,\ \pi_2(\hat{\tau}) \rangle$$
$$\tau_{\mathit{call}}(\langle S, \_, \_, \hat{\tau} \rangle, \langle \hat{\tau}_c, \_, \lambda \mathbf{y} \,.\, S_c \rangle) = \langle S_c^\ell,\ \hat{\tau}' \rangle$$

$$\text{where } \hat{\tau}' = \begin{cases} \textbf{first } k \textbf{ of } (\textbf{self} :: \pi_2(\hat{\tau})) & \text{if } S \in \mathit{UserCalls} \\ \pi_2(\hat{\tau}_c) & \text{if } S \in \mathit{ReturnKont} \\ \pi_2(\hat{\tau}) & \text{otherwise} \end{cases}$$

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).

Property simulation analysis A more unusual form of control-flow sensitivity is Das et al.'s property simulation [11]. 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 $\Theta^\sharp$ 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

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