Full Control-Flow Sensitivity for Definitional Interpreters

Kimball Germane

Brigham Young University

Abstract. Static analyzers can exhibit a variety of control-flow sensitivities, including path and flow sensitivity. Darais *et al.* provide an account of these sensitivities rooted in "control properties of the interpreter" for static analyzers that model program behavior as a finite-state transition system. In the meantime, many static analyzer frameworks—particularly those for higher-order languages—have migrated to more sophisticated and precise pushdown models which admit evaluation summaries. It is not immediately clear how to realize the full spectrum of path and flow sensitivity in a summary-based setting which, like that of Darais *et al.*, is rooted in the control properties of the interpreter and therefore independent of all other aspects of the analyzer formulation. We present a framework which achieves precisely this. We also provide a caching algorithm which performs summarization and demonstrate the framework on an abstract definitional interpreter. Altogether, we show how to achieve the full range of path and flow sensitivities, even at once, within a single abstract definitional interpreter-based analysis, completely independent of other aspects of its formulation.

1 Control-Flow Sensitivity

A central feature of a program analysis is its *control-flow sensitivity* [15], or how precisely it accounts for the program's realizable execution paths. Types of control-flow sensitivity include path sensitivity, which maintains high fidelity to realizable execution paths, and flow sensitivity, which aggregates the contributions of paths as they pass through each program point.

Modular abstract interpreters [6] account for path and flow sensitivity through the associativity of each fact base to individual program points in the system space of the analysis. We recap this account using the program in Figure 1. In this program, the `if0` construct produces the value of its first block if its discriminee is zero and the value of its second block if it is nonzero. We assume that the zero–nonzero status of N is not determined when control enters the program. We now break down how different associativities between fact bases and program points induce different control-flow sensitivities within an analysis.

1. *Path-sensitive analysis* A path-sensitive analysis tracks and correlates data and control flow precisely by distinguishing the occurrence of each fact base and program point. In such an analysis, program points 3 and 4 are are

```
1: let x :=                          | in
2:   if0(N){                         | 5:   let y :=
3:     if0(N){1} else {2}            | 6:     if0(N){5} else {6}
     } else {                        |      in
4:     if0(N){3} else {4}            | 7:     exit(x, y)
     }                               |
```

Fig. 1. A program to illustrate path and flow sensitivity

mutually exclusive within paths as the analysis holds $\{N = 0\}$ at program point 3 and $\{N \neq 0\}$ at program point 4. This distinction is maintained so that the analysis arrives at program point 6 along two control flow paths. In one, the analysis holds the fact base $\{N = 0, x = 1\}$ and, in the other, $\{N \neq 0, x = 4\}$. By program point 7, these fact bases are respectively $\{N = 0, x = 1, y = 5\}$ and $\{N \neq 0, x = 4, y = 6\}$ so that the analysis has perfectly correlated the data flow of x and y.

2. *Flow-sensitive analysis* A flow-sensitive (but path-insensitive) analysis associates a fact base with each program execution point but does not preserve path-sensitive facts obtained up to that point. As before, such an analysis holds $\{N = 0\}$ at program point 3 and $\{N \neq 0\}$ at program point 4, and is thus able to evaluate only one side of each conditional. However, at program point 5, these fact bases are merged to $\{N \in \mathbb{Z}, x \in \{1, 4\}\}$. Having lost a precise account of N, the analysis explores each branch of the conditional at program point 6 and, at program point 7, holds $\{N \in \mathbb{Z}, x \in \{1, 4\}, y \in \{5, 6\}\}$, which does not correlate `x` and `y`.
3. *Flow-insensitive analysis* A flow-insensitive analysis associates a single fact base with all program points collectively which contains facts which must hold at every program point. Because the most precise fact about N that holds at every program point is $N \in \mathbb{Z}$, the analysis must consider every branch of every conditional guarded on N. At program point 7, its fact base is $\{N \in \mathbb{Z}, x \in \{1, 2, 3, 4\}, y \in \{5, 6\}\}$.

Modular abstract interpreters earn the descriptor *modular* in part by allowing the associativity between fact bases and program points to be treated independently of other aspects of the analysis design and implementation. Because it is regulated by this associativity, the control-flow sensitivity of the analysis is then independent of those aspects as well.

1.1 Control-Flow Sensitivity in Pushdown Models

The preceding discussion proceeded in terms of program *points*, the primary constituents of the finite-state models produced by the underlying (modular) abstract interpreter. Many modern analysis frameworks—particularly those for higher-order languages—model control behavior as a pushdown system computed using summarization [3]. These frameworks therefore produce summaries

which offer a richer account of control behavior than mere program points. It is not immediately clear in this setting whether it is possible and how to realize path and flow sensitivity in a way both reflected as particular associativities in the system space and independent of other analysis aspects.

In this paper, we present a framework that achieves precisely this: the full gamut of control-flow sensitivity for pushdown models facilitated by system space associativities (§4) and alterable independent of other analysis design and implementation aspects (§5). Our framework also encompasses a unified system space which allows each kind of sensitivity to manifest at once in a single analysis.

We present our framework in the setting of definitional interpreters which both is consistent with modern frameworks [5, 26] and cleanly separates the language semantics from the control-flow sensitivity. We start by defining such an interpreter written in a particular monadic style, allowing it to be parameterized over several key aspects of the analysis (§2). We then instantiate this interpreter abstractly in steps before (§3) and after (§7) we present our framework.

The setting in which we instantiate our framework resembles that of the popular *Abstracting Definitional Interpreters* (ADI) [5] framework. The ADI framework uses monads and monad transformers to allow one to swiftly construct and modify an analysis. Through the interactions between these monad transformers, it offers a form of widening we term *result widening* which exhibits aspects of both path and flow sensitivity. We show how to integrate result widening into our framework, thereby subsuming the control-flow sensitivity expressiveness of the ADI framework (§8).

Summarization algorithms are intricate and caching algorithms can be surprisingly subtle. As part of our framework, we also present a caching algorithm to compute an analysis (§6).

We conclude by discussing related (§10) and future work (§11).

2 A Definitional Interpreter

To ground the presentation of our framework, we introduce a language by way of a definitional interpreter, which will offer us a concrete artifact to embed within our framework.

2.1 Language

We implement a definitional interpreter for the language λ**IF** [6], the λ calculus extended with integer operations and conditionals. Our framework can smoothly handle semantics with mutation and strong update [2] via, e.g., abstract counting [20, 16], recursive functions, compound data structures via, e.g., algebraic data types, and other features. The language λ**IF** is sufficient to demonstrate different sensitivities within the framework.

The grammar of λ**IF** is as follows.

$$e \in \mathbf{Exp} ::= a \mid e \odot e \mid \mathsf{if0}(e)\{e\}\,\mathsf{else}\,\{e\} \qquad a \in \mathbf{Atom} ::= i \mid x \mid \lambda(x).e$$

$$i \in \mathbb{Z} \qquad x \in \mathbf{Var} \qquad \odot \in \mathbf{Op} ::= \oplus \mid @ \qquad \oplus \in \mathbf{IOp} ::= + \mid -$$

A program is a closed expression in **Exp**, which includes atomic expressions, binary operator applications, and a conditional construct. An atom in **Atom** is either an integer, a variable reference, or a λ term. Binary operators are either integer operators for arithmetic or function application. In addition to these domains, we make use of the set of λs **Lam** when defining closures and the set of application sites **Call** when defining timestamps.

2.2 State Space

The forthcoming definitional interpreter uses the following state space definitions.

$$\alpha \in \mathit{Addr}^{\mathit{Time}} := \mathbf{Var} \times \mathit{Time} \qquad \rho \in \mathit{Env}^{\mathit{Time}} := \mathbf{Var} \rightharpoonup \mathit{Addr}^{\mathit{Time}}$$
$$\sigma \in \mathit{Store}^{\mathit{Time},\mathit{Val}} := \mathit{Addr}^{\mathit{Time}} \to \mathit{Val} \qquad c \in \mathit{Clo}^{\mathit{Time}} := \mathbf{Lam} \times \mathit{Env}^{\mathit{Time}}$$

These definitions are parameterized over *Time* and *Val* the instantiations of which will vary with context. The structure of addressses is fixed as a pair of a variable and a timestamp. From the structure of addresses follows that of environments as finite maps from variables to addresses and stores as maps from addresses to values. In each semantics the set of values will be a domain (equipped with a partial order and bottom element) that includes the set of closures.

2.3 The Interpreter

Following Darais *et al.* [6], the interpreter itself is written in a particular monadic style and is parameterized by:

1. a value domain which comprises the abstraction of primitive values and the fidelity of primitive operations;
2. a timestamp discipline, which determines the kind of context sensitivity (e.g. call-site or object sensitivity), as well as its degree; and
3. a monad which mediates access to analyzer functionality through various effects (e.g. state and nondeterminism).

That is, we define the interpreter once and for all and instantiate it by defining each of these aspects. We do so to recover an abstract semantics in §3 and a concrete semantics in an accompanying technical report [8].

Value Domain The value domain comprises an abstract domain *Val*, injection and projection operations for *Val*, and a denotation for primitive operators $\delta[\![\cdot]\!]$.

Val must be a join-semilattice with

$$\bot : \mathit{Val} \qquad \cdot \sqcup \cdot : \mathit{Val} \times \mathit{Val} \to \mathit{Val}$$

that respect the usual join-semilattice laws.

We must also provide operations which inject integers and closures into the value domain and operations which project values into finite observations of integers or closures.

$$I_{int} : \mathbb{Z} \to \mathit{Val} \qquad E_{if0} : \mathit{Val} \to \mathcal{P}(\mathit{Bool})$$
$$I_{clo} : \mathit{Clo}^{\mathit{Time}} \to \mathit{Val} \qquad E_{clo} : \mathit{Val} \to \mathcal{P}(\mathit{Clo}^{\mathit{Time}})$$

These operations should obey the following adjoint laws.

$$\{\mathsf{true}\} \subseteq E_{if0}(I_{int}(i)) \text{ if } i = 0$$
$$\{\mathsf{false}\} \subseteq E_{if0}(I_{int}(i)) \text{ if } i \neq 0$$
$$\{c\} \subseteq E_{clo}(I_{clo}(c))$$

Finally, primitive operation interpretation via $\delta[\![\cdot]\!]$ must be sound with respect to the value refinement relation.

$$I_{int}(i_0 + i_1) \sqsubseteq \delta[\![+]\!](I_{int}(i_0), I_{int}(i_1))$$
$$I_{int}(i_0 - i_1) \sqsubseteq \delta[\![-]\!](I_{int}(i_0), I_{int}(i_1))$$

Abstract Time A timestamp embedded within configurations can be used to effect various forms of context sensitivity [24, 10], including call-site sensitivity [22] and object sensitivity [23]. Whatever its kind, this sensitivity is orthogonal to the analysis's path and flow sensitivity. We demonstrate in this paper call-site sensitivity, instantiating *Time* as a sequence of call sites, bounded to a length provided as a parameter k to the analysis. Even after selecting call-site sensitivity, however, we are able to realize different variants of it by treating it as a per-path component or a path-sensitive component, which we discuss later in this section.

In our setting, an abstract time is determined by a set of times *Time* and an operation $\mathit{tick} : \mathbf{Call} \times \mathit{Time} \to \mathit{Time}$. There are no laws that *tick* must obey; instead, the set of times merely provides unique markers which control the precision at which analysis proceeds.

The Monad A monad $m : \mathit{Type} \to \mathit{Type}$ comprises a type operator and two associated operations.

$$\mathit{return} : \forall A.A \to m(A) \qquad \mathit{bind} : \forall A.\forall B.m(A) \to (A \to m(B)) \to m(B)$$

In general, we use the standard semicolon notation $\mathbf{do}\, x \leftarrow c; f(x)$ in place of $\mathit{bind}(c)(f)$ and allow newlines in place of semicolons in multi-line definitions headed by **do**. Such a monad implements a reader effect for a type r if it includes the operators

$$\mathit{ask} : m(r) \qquad \mathit{inEnv} : \forall A.r \times m(A) \to m(A)$$

where *ask* obtains the environment value and *inEnv* installs a given environment value for a given computation. Such a monad implements a state effect for a type s if it includes operators

$$get : m(s) \qquad\qquad put : s \to m(1)$$

where *get* produces the state, *put* sets it, and both satisfy the state laws [9] Such a monad implements a nondeterminism effect if it includes operators

$$\cdot\langle+\rangle\cdot : \forall A.m(A) \times m(A) \to m(A) \qquad\qquad mzero : \forall A.m(A)$$

where, for this work, $m(A)$ is a join-semilattice, with join operator $\langle+\rangle$ and bottom element *mzero*. Finally, such a monad implements a writer effect for a type w if it includes an operator

$$log : w \to m(1).$$

Evaluator Figure 2 presents a definitional interpreter for **λIF** parameterized by a given a value domain, timestamp discipline, and monad, each as described above. The evaluator dispatches on the variant of expression being evaluated and

$$\begin{array}{l}
eval^m : (\mathbf{Exp} \to m(\mathit{Val})) \to \mathbf{Exp} \to m(\mathit{Val}) \\
eval^m(\mathit{eval})(e) := \\
\quad \mathbf{case}\, e\, \mathbf{of} \\
\quad\quad a \Rightarrow \mathcal{A}^m[\![a]\!] \\
\quad\quad e_0 \oplus e_1 \Rightarrow \mathbf{do}^m\, v_0 \leftarrow \mathit{eval}(e_0); v_1 \leftarrow \mathit{eval}(e_1); \mathit{return}^m(\delta[\![\oplus]\!](v_0, v_1)) \\
\quad\quad e_0\, @\, e_1 \Rightarrow \mathbf{do}^m\, v_0 \leftarrow \mathit{eval}(e_0); v_1 \leftarrow \mathit{eval}(e_1); (\lambda(x).e', \rho) \leftarrow \uparrow_m(E_{clo}(v_0)) \\
\quad\quad\quad \mathit{tick}^m(e, \mathbf{do}^m\, \alpha \leftarrow \mathit{alloc}^m_\sigma(x); \mathit{extend}^m_\sigma(\alpha, v_1); \mathit{inEnv}_\rho(\rho[x \mapsto \alpha], \mathit{eval}(e'))) \\
\quad\quad \mathsf{if0}(e_0)\{e_1\}\,\mathsf{else}\,\{e_2\} \Rightarrow \mathbf{do}^m\, v_0 \leftarrow \mathit{eval}(e_0); b \leftarrow \uparrow_m(E_{if0}(v_0)); \mathit{refine}^m(e_0, b) \\
\quad\quad\quad \mathit{eval}(\mathsf{ite}(b, e_1, e_2))
\end{array}$$

Fig. 2. Definitional interpreter for **λIF**

behaves accordingly. To evaluate an atomic expression a, the evaluator appeals to $\mathcal{A}^m$, defined below. To evaluate a binary expression $e_0 \oplus e_1$, the evaluator evaluates the left argument, then the right, and then uses the primitive operation interpretation metafunction $\delta[\![\cdot]\!]$ of the given value domain to carry out the actual operation. Apparently-recursive calls in this and subsequent clauses of the evaluator are intercepted by the functional argument *eval*; defining the evaluator in this open-recursive style allows it to be instrumented, which we demonstrate in §3. To evaluate an application expression $e_0@e_1$, the evaluator

evaluates the function, then the argument, and then uses $\uparrow_m$, defined below, to extract a closure projected from the function value by E_{clo}. With a closure in hand, the evaluator uses $tick^m$ to increment the timetamp, $alloc^m_\sigma$ to allocate a new address, and $extend^m_\sigma$ to bind the argument in the store, each defined below. Finally, to evaluate a conditional expression $\mathsf{if0}(e_0)\{e_1\}\,\mathsf{else}\,\{e_2\}$, the evaluator evaluates the guard expression, then uses $\uparrow_m$ to extract a boolean projected from the guard value by E_{if0}. Before using this value to choose the appropriate branch, $refine^m$, defined below, narrows the guard value when the guard expression is merely a reference to it. The entire evaluator is parameterized by a monad m, assumed to provide reader, nondeterminism, and state effects. Each primitive superscripted by m is assumed to have access to these effects as well.

Primitive Evaluation Atomic expression evaluation is carried out by $\mathcal{A}^m$, defined in Figure 3. Evaluation of an integer i returns the abstract value produced

$$
\begin{aligned}
&\mathcal{A}^m[\![\cdot]\!] : \mathbf{Atom} \to m(\mathit{Val}) \\
&\mathcal{A}^m[\![i]\!] = \mathit{return}^m(I_{int}(i)) \\
&\mathcal{A}^m[\![x]\!] = \mathbf{do}^m\ \rho \leftarrow \mathit{ask}^m_\rho; \mathsf{if}\ x \in \rho\ \mathsf{then}\ \mathit{lookup}^m_\sigma(\rho(x))\ \mathsf{else}\ \mathit{return}^m(\bot) \\
&\mathcal{A}^m[\![\lambda(x).e]\!] = \mathbf{do}^m\ \rho \leftarrow \mathit{get}^m_\rho; \mathit{return}^m(I_{clo}(\lambda(x).e, \rho))
\end{aligned}
$$

Fig. 3. The atomic evaluation function

by the integer injection function I_{int}. Evaluation of a reference x obtains the environment and uses $lookup^m_\sigma$ to look up its address in the store if it is in scope and returns the bottom value if not. Evaluation of a λ obtains the environment and returns the abstract value produced by the closure injection function I_{clo}.

Context Sensitivity We hardwire call-site sensitivity into this interpreter by placing the $tick^m$ function, responsible for incrementing the timestamp, at each call. Its definition is in terms of a reader effect, which yields the sensitivity of stack-based k-CFA (and m-CFA) in which the timestamp captures the top-k stack frames. Under this effect, timestamps are treated with the same stack discipline as environments.

$$
\begin{aligned}
&\mathit{tick}^m : \forall A.\mathbf{Call} \times m(A) \to m(A) \\
&\mathit{tick}^m(e, \mathit{cmp}) := \mathbf{do}^m\ \tau \leftarrow \mathit{ask}^m_\tau; \mathit{inEnv}^m_\tau(\mathit{tick}(e, \tau), \mathit{cmp})
\end{aligned}
$$

If the timestamp is managed instead by a state effect, as follows, the resulting sensitivity is instead that of traditional k-CFA in which the timestamp captures the most-recent-k calls. Under this effect, timestamps are treated with the same threading discipline as stores.

$$
\begin{aligned}
&\mathit{tick}^m_{alt} : \forall A.\mathbf{Call} \times m(A) \to m(A) \\
&\mathit{tick}^m_{alt}(e, \mathit{cmp}) := \mathbf{do}^m\ \tau \leftarrow \mathit{get}^m_\tau; \mathit{put}^m_\tau(\mathit{tick}(e, \tau)); \mathit{cmp}
\end{aligned}
$$

Value Refinement $refine^m$ allows evaluation in a branch to remember which branch was taken, provided that the branch guard is syntactically a variable.

$$
\begin{aligned}
&refine^m : \mathbf{Exp} \times Bool \to m(1) \\
&refine^m(x, b) := \mathbf{do}^m\, \rho \leftarrow get^m_\rho; \sigma \leftarrow get^m_\sigma; \\
&\qquad\qquad \mathsf{let}\, \alpha := \rho(x)\, \mathsf{in}\, put_\sigma(\sigma[\alpha \mapsto \sigma(\alpha) \sqcap f(b)]) \\
&refine^m(\_, b) := return^m(\langle\rangle)
\end{aligned}
$$

where

$$
f(b) := \begin{cases} \sqcup\{I_{int}(i) : i \in \mathbb{Z}, i = 0\} \text{ if } b = \mathsf{true} \\ \sqcup\{I_{int}(i) : i \in \mathbb{Z}, i \neq 0\} \text{ if } b = \mathsf{false} \end{cases}
$$

It increases precision when the refined value is treated flow-sensitively. To illustrate, suppose in the evaluation of $\mathsf{if0}(x)\{x + 1\}\,\mathsf{else}\,\{1\}$ that x is bound at α in the environment which is itself bound to v in the store. Further suppose that the integer projection of v $E_{if0}(v) = \{0, 1\}$ so that the evaluator evaluates both branches. Without $refine^m$, the evaluator will produce a value v_r from the consequent branch such that $I_{int}(0 + 1) \sqcup I_{int}(1 + 1) \sqsubseteq v_r$. With $refine^m$, however, the evaluator sharpens v in the store before branch evaluation such that $E_{if0}(v) = \{0\}$ so that $I_{int}(0 + 1) \sqsubseteq v_r$.

Store Access Access to the store is abstracted behind $alloc^m$, $lookup^m$, and $extend^m$ operations, defined themselves directly in terms of monadic state effects as follows.

$$
\begin{aligned}
&alloc^m : \mathbf{Var} \to m(Addr) \\
&alloc^m(x) := \mathbf{do}^m\, \tau \leftarrow ask^m_\tau; return(x, \tau) \\
&lookup^m : Addr \to m(\,Val) \\
&lookup^m(\alpha) := \mathbf{do}\, \sigma \leftarrow get_\sigma; return(\sigma(\alpha)) \\
&extend^m : Addr \times Val \to m(1) \\
&extend^m(\alpha, v) := \mathbf{do}^m\, \sigma \leftarrow get^m_\sigma; put^m_\sigma(\sigma \sqcup [\alpha \mapsto v])
\end{aligned}
$$

Reflecting Nondeterminism The $\uparrow_m$ reflects a set of values as a nondeterministic computation within the monad m.

$$
\begin{aligned}
&\uparrow_m : \forall A.\mathcal{P}(A) \to m(A) \\
&\uparrow_m (\{x_1, \ldots, x_n\}) := return^m(x_1)\langle+\rangle^m \ldots \langle+\rangle^m return^m(x_n)
\end{aligned}
$$

Within the evaluator, it is used exclusively to reflect a set of results produced by a projection function E_{clo} and E_{if0} as a nondeterministic result within the monadic computation.

3 Recovering an Abstract Semantics

To recover an abstract semantics, we once again instantiate the three parameters of the interpreter: the value domain, the timestamp set, and the underlying

$$\widehat{Val} := \mathcal{P}(\{+, 0, -\} \cup \widehat{Clo}) \qquad \widehat{Time} := \mathbf{Call}^{\leq k} \qquad \widehat{Clo} := Clo^{\widehat{Time}}$$

$$\widehat{Env} := Env^{\widehat{Time}} \qquad \widehat{Store} := Store^{\widehat{Time}, \widehat{Val}}$$

$$I_{int} : \mathbb{Z} \to \widehat{Val} \qquad E_{if0} : \widehat{Val} \to \mathcal{P}(Bool)$$

$$I_{int}(i) := \begin{cases} \{-\} \text{ if } i < 0 \\ \{0\} \text{ if } i = 0 \\ \{+\} \text{ if } i > 0 \end{cases} \qquad E_{if0}(v) := \{\mathsf{true} : 0 \in v\} \cup \{\mathsf{false} : - \in v \vee + \in v\}$$

$$I_{clo} : Clo^{\widehat{Time}} \to \widehat{Val} \qquad E_{clo}(v) := \{c : c \in v\}$$

$$I_{clo}(c) = \{c\} \qquad E_{clo} : \widehat{Val} \to \mathcal{P}(Clo^{\widehat{Time}})$$

$$tick : \mathbf{Call} \times Time \to Time \qquad \delta : \mathbf{IOp} \to \widehat{Val} \times \widehat{Val} \to \widehat{Val}$$

$$tick(e, \tau) := \lfloor e\tau \rfloor_k \qquad \delta[\![+]\!](v_0, v_1) := \{i_0 + i_1 : i_0 \in v_0; i_1 \in v_1\}$$

$$\delta[\![-]\!](v_0, v_1) := \{i_0 - i_1 : i_0 \in v_0; i_1 \in v_1\}$$

Fig. 4. Value and time definitions for an abstract semantics

monad. Figure 4 presents the value and time abstractions. Following Darais *et al.* [6], we use the sign abstraction for integers. To obtain a finite set of times, we use the standard technique of limiting each time to at-most k call sites [22]. Because addresses are derived from times, this limiting has the effect of finitizing the address space as well. Thus, it becomes possible (and probable) that the allocator will be able to produce only addresses that are already in use. To maintain soundness, the *Abstracting Abstract Machines* [24] methodology (and its successor ADI [5]) stipulates that new data allocated at these addresses are joined with the data already residing there. When these addresses are dereferenced, the returned data must subsume both the old and new data. Hence, we use a powerset representation in the abstract semantics to model nondeterminism.

What remains is to define a monad the abstract semantics with appropriate effects. However, the control-flow sensitivity exhibited by the analyzer depends on this definition. Hence, we will refrain from defining it until after introducing our framework.

4 Path and Flow Sensitivity in the System Space

4.1 From Small-Step...

Darais *et al.* [6] observe that different control flow sensitivities induce different system spaces of analysis. A path-sensitive analysis allows arbitrary relationships between program points *Point* and fact bases B, yielding a system space of

$\mathcal{P}(\mathit{Point} \times B)$. On the other hand, a path- and flow-insensitive analysis associates a fact base B with all of the program points, yielding a system space $\mathcal{P}(\mathit{Point}) \times B$. Finally, a path-insensitive but flow-sensitive analysis associates a fact base B with each program point, yielding a system space $\mathit{Point} \mapsto B$.

Although apparently point-centric, these system spaces capture evaluation paths through their constituent points implicitly: when each contained *Point* is paired with its corresponding fact base, its successor is determined by the transition relation, allowing one to reconstruct paths step by step.

Darais *et al.* use the observed connection between control-flow sensitivity and the shape of the system space to systematize control-flow sensitivity within a framework based on Galois transformers. A *Galois transformer* (GT) is a transformer which augments the state space with a fact base paired with a discipline of that fact base which induces a particular control-flow sensitivity. Galois transformers have been dispatched to construct abstract machine-based static analyzers whose control-flow sensitivity—and supporting system space—varies with the particular Galois transformers used to build the analysis, and the order in which they are applied.

4.2 ...to Big-Step

The shape of the system spaces produced by the GT framework is in large part an artifact of it being built around small-step semantics. In such settings, program behavior is typically modelled as a transition system of program states and a finitization is typically used to ensure that an abstraction of that model is computable [24]. Since the introduction of the GT framework, several frameworks have been developed which utilize big-step semantics, which allows static analyzers to be formulated in terms of definitional interpreters. Rather than being point- or state-centric, definitional interpreter-based static analysis is summary-centric, with system spaces which resemble $\mathit{Config} \mapsto \mathit{Result}$ and record the behavior of the stack-agnostic evaluation of each configuration *Config*.

It is not immediately clear how to realize each flavor of control-flow sensitivity in this setting, nor how each sensitivity is reflected in the system space. We now explore both in turn, first determining how the different flow sensitivities manifest in this setting and then devising a state space that can accommodate that meaning.

To determine how the flow sensitivities manifest, we return to the program in Figure 1. However, instead of conceptualizing the execution of this program as a path from point to point, we view it as the evaluation of labelled expressions and subexpressions. In this view, the overall program is the let expression spanning labels 1–7. Its bound expression spans labels 2–4 and its body, another let expression, spans labels 5–7. We now consider each kind of control-flow sensitivity in turn and synthesize a system space which accommodates it.

Path-sensitive analysis Figure 5 presents a path-sensitive, summary-based analysis of the program. Each entry in the table consists of a summary which comprises the fact base at the commencement of the expression's evaluation and the resultant value and fact base at its conclusion. Evaluation begins at label 1 with

Expression	Path 1	Path 2
1	$\{N \in \mathbb{Z}\}$ $\langle (1,5), \{N = 0, x = 1, y = 5\}\rangle$	$\{N \in \mathbb{Z}\}$ $\langle (4,6), \{N \neq 0, x = 4, y = 6\}\rangle$
2	$\{N \in \mathbb{Z}\}$ $\langle 1, \{N = 0\}\rangle$	$\{N \in \mathbb{Z}\}$ $\langle 4, \{N \neq 0\}\rangle$
3	$\{N = 0\}$ $\langle 1, \{N = 0\}\rangle$	
4		$\{N \neq 0\}$ $\langle 4, \{N \neq 0\}\rangle$
5	$\{N = 0, x = 1\}$ $\langle (1,5), \{N = 0, x = 1, y = 5\}\rangle$	$\{N \neq 0, x = 4\}$ $\langle (4,6), \{N \neq 0, x = 4, y = 6\}\rangle$
6	$\{N = 0, x = 1\}$ $\langle 5, \{N = 0, x = 1\}\rangle$	$\{N \neq 0, x = 4\}$ $\langle 6, \{N \neq 0, x = 4\}\rangle$
7	$\{N = 0, x = 1, y = 5\}$ $\langle (1,5), \{N = 0, x = 1, y = 5\}\rangle$	$\{N \neq 0, x = 4, y = 6\}$ $\langle (4,6), \{N \neq 0, x = 4, y = 6\}\rangle$

Fig. 5. Path-sensitive, summary-based analysis

an effectively-empty fact base, as does the evaluation of the first bound expression at label 2. The evaluation of the guard at label 2 splits the world, creating a path for each outcome of its evaluation. The first path reaches label 3 with the fact base $\{N = 0\}$ and the guard holds. The consequent expression 1 is trivially evaluated and 1 is returned from the conditional at label 3 and the containing conditional at label 2. Crucially, the fact base itself, which encodes the path travelled through evaluation, is returned with the result. Consequently, the fact base of the evaluation of the body expression at label 5 is $\{N = 0, x = 1\}$. This then is the fact base at the evaluation of the expression at label 6. Again, the guard holds, and 5 is produced along with the fact base. At the evaluation of the body expression at label 7, the fact base is $\{N = 0, x = 1, y = 5\}$. The evaluation of the second path branching from label 2 proceeds similarly, and results in the fact base $\{N \neq 0, x = 4, y = 6\}$. These fact bases are identical to the ones produced in the path-sensitive, point-centric analysis of the same program, as we would expect. The key is that the fact base must be threaded through evaluation, being included in each evaluation configuration and each produced result. This observation leads to a system space $\mathit{Config} \times B \mapsto \mathcal{P}(\mathit{Val} \times B)$.

Flow-insensitive analysis A flow-insensitive analysis records within the fact base facts that are true at every program point. Thus, the fact base is not associated with any particular configuration or result, but all configurations and results. For such an analysis, rather than proceed through evaluation sequentially, it makes more sense here to consider the contribution that each part of the program makes to the global fact base, given the contributions of the other parts. For example, the evaluation of the guard at label 2 considers both outcomes and records each in the same global fact base, which concludes merely $\{N \in \mathbb{Z}\}$. Under such a coarse approximation, the evaluations of the conditionals at labels 3 and 4 evaluate both branches, as does the evaluation of the conditional at label 6. As in the point-

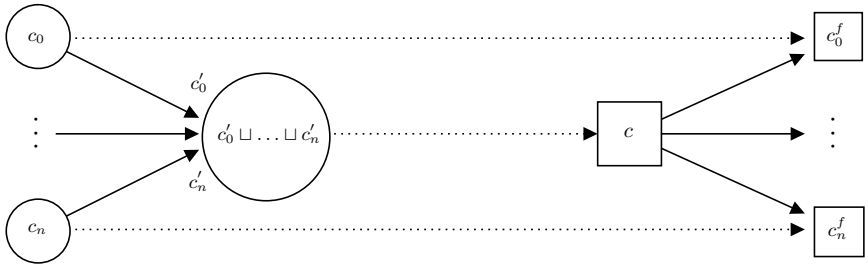

Fig. 6. Flow-sensitive quantities in a summary-based analysis

centric analysis, the final fact base is $\{N \in \mathbb{Z}, x \in \{1,2,3,4\}, y \in \{5,6\}\}$, and the system space supporting this sensitivity is $[\mathit{Config} \mapsto \mathcal{P}(\mathit{Val})] \times B$ with a single fact base shared by the entire evaluation space.

Flow-sensitive analysis A flow-sensitive (but path-insensitive) analysis produces a solution for every program point $[14, 13, 11, 15, 4]$. How do we effect this in a setting that is summary- rather than point-centric? The idea is that flow sensitivity with respect to the fact base occurs when all paths through a particular program point contribute to that quantity. A summary encapsulates a path segment—not a point—so we apply this idea to say that treatment of a fact base is flow-sensitive when all paths through each path segment contribute to that quantity along the segment. Because a summary provides a snapshot of the beginning and the end of the segment, we record the fact base at the beginning and end. That is, each configuration in the program is associated with *two* fact bases, one recording the contribution of paths as they enter its evaluation and another recording the contribution of paths as they exit. Figure 6 illustrates this idea. The circular nodes represent configurations and the square nodes results. If we have a set of configurations with flow-sensitive quantities $c_0, \ldots, c_n$ which, after some evaluation, arrive at some configuration with flow-sensitive quantities $c'_0, \ldots, c'_n$, then the flow-sensitive quantity at that configuration is their join $c'_0 \sqcup \cdots \sqcup c'_n$. If that configuration yields a result with flow-sensitive quantity c, this quantity is propagated to each of the original evaluation paths which, after some evaluation, yield results with flow-sensitive quantities $c^f_0, \ldots, c^f_n$, respectively. Each configuration is associated with its constituent flow-sensitive quantity and that of its result, identified by dashed edges in the diagram.

Using this idea, we now illustrate a flow-sensitive, summary-centric analysis of the program in Figure 7. At the initial program configuration, the sole path has an effectively empty fact base. This fact base is in effect as that path enters evaluation of the let body on lines 2–4. The evaluation of the guard splits the world so that the evaluation of the conditionals on lines 3 and 4 have entry fact bases of $\{N = 0\}$ and $\{N \neq 0\}$, respectively. The evaluation of label 3's guard holds and its consequent expression 1 is evaluated with an entry and exit fact base of $\{N = 0\}$, which is also the result of the overall conditional. The evaluation of the conditional at label 4 proceeds similarly, as it yields 4 with

Expression	Entry	Exit
1	$\{N \in \mathbb{Z}\}$	$\langle\{(1,5),(1,6),(4,5),(4,6)\}, \{N \in \mathbb{Z}, x \in \{1,4\}, y \in \{5,6\}\}\rangle$
2	$\{N \in \mathbb{Z}\}$	$\langle\{1,4\}, \{N \in \mathbb{Z}\}\rangle$
3	$\{N = 0\}$	$\langle\{1\}, \{N = 0\}\rangle$
4	$\{N \neq 0\}$	$\langle\{4\}, \{N \neq 0\}\rangle$
5	$\{N \in \mathbb{Z}, x \in \{1,4\}\}$	$\langle\{(1,5),(1,6),(4,5),(4,6)\}, \{N \in \mathbb{Z}, x \in \{1,4\}, y \in \{5,6\}\}\rangle$
6	$\{N \in \mathbb{Z}, x \in \{1,4\}$	$\langle\{5,6\}, \{N \in \mathbb{Z}, x \in \{1,4\}\}$
7	$\{N \in \mathbb{Z}, x \in \{1,4\}, y \in \{5,6\}\}$	$\langle\{(1,5),(1,6),(4,5),(4,6)\}, \{N \in \mathbb{Z}, x \in \{1,4\}, y \in \{5,6\}\}\rangle$

Fig. 7. A flow-sensitive, summary-centric analysis

a fact base of $\{N \neq 0\}$. Now evaluation reaches a join point of these paths as it exits the evaluation of the containing let expression and the resulting values are joined to $\{1,4\}$ and resulting fact bases to $\{N \in \mathbb{Z}\}$. Evaluation of the let expression on at label 5 then proceeds with the fact base $\{N \in \mathbb{Z}, x \in \{1,4\}\}$. Although evaluation of each branch of the conditional at label 6 occurs in a fact base with refined knowledge of N, the branch results are joined so that the overall conditional produces the values $\{5,6\}$ and the fact base $\{N \in \mathbb{Z}, x \in \{1,4\}\}$. The evaluation of the expression at label 7 then commences and completes with the fact base $\{N \in \mathbb{Z}, x \in \{1,4\}, y \in \{5,6\}\}$, precisely the fact base we obtained in the flow-sensitive, point-centric analysis. The key to achieving flow sensitivity in a summary-centric analysis was to maintain entry and exit fact bases for each configuration evaluation, leading to the system space

$$[\mathit{Config} \mapsto B] \times [\mathit{Config} \mapsto \mathcal{P}(\mathit{Val}) \times B].$$

5 Our Framework: Full Path and Flow Sensitivity in Abstract Definitional Interpreters

Bulding on the intuition developed in the previous section, we now introduce our framework for full path and flow sensitivity within summary-centric analysis.

5.1 State Spaces

We begin by describing the system space for each control-flow sensitivity and then present a unified system space that exhibits all of the sensitivities. The system spaces are parameterized by four types, U, V, W, and Y, which serve the following roles:

1. U is a per-path component capturing an aspect of the continuation. That is, it does not distinguish different paths which coincide from the point of capture. Examples of such components are environments, stack-based k-CFA/m-

CFA-style context sensitivity [21], and stack address sets for abstract garbage collection. We require that U is finite to ensure computability.
2. V is path-sensitive, distinguishing the path that was taken. Examples at this sensitivity are per-configuration stores and traditional k-CFA-style context sensitivity. We also require that V is finite to ensure computability.
3. W is flow-sensitive, capturing what is true of every path through a segment of evaluation at the beginning and end of the segment. An example at this sensitivity is flow-sensitive stores (a form of configuration widening [19]). We require that W be a join-semilattice with no infinite ascending chains.
4. Y is flow-insensitive, capturing what is true of all paths at all points in evaluation. An example at this sensitivity is a globally-widened store. We also require that Y be a join-semilattice with no infinite ascending chains.

Because the type U is invariably associated with the control expression within configurations, we define $\mathit{Control} := \mathbf{Exp} \times U$, which will feature in each system space. Additionally, each system space includes a set of reachable *Control*s (possibly attached to a path-sensitive quantity).

Path-sensitive system space The system space

$$\Sigma_{ps} := \mathcal{P}(\mathit{Control} \times V) \times [\mathit{Control} \times V \mapsto \mathcal{P}(\mathit{Val} \times V)]$$

supports **p**ath-**s**ensitive treatment of V.

Flow-sensitive system space For a join-semilattice $(W, \sqsubseteq_W)$, the system space

$$\Sigma_{fs} := \mathcal{P}(\mathit{Control}) \times [\mathit{Control} \mapsto W] \times [\mathit{Control} \mapsto \mathcal{P}(\mathit{Val}) \times W]$$

supports **f**low-**s**ensitive, path-insensitive treatment of W.

Flow-insensitive system space For a join-semilattice $(Y, \sqsubseteq_Y)$, the system space

$$\Sigma_{fi} := \mathcal{P}(\mathit{Control}) \times [\mathit{Control} \mapsto \mathcal{P}(\mathit{Val})] \times Y$$

supports **f**low-**i**nsensitive treatment of Y.

The unified system space

$$\Sigma := \mathcal{P}(\mathit{Control} \times V) \times [\mathit{Control} \times V \mapsto W] \times [\mathit{Control} \times V \mapsto \mathcal{P}(\mathit{Val} \times V) \times W] \times Y$$

realizes all flow sensitivities in their respective types at once. Some components, such as fact bases, can be distributed across these flow sensitivities within a single analysis.

Hereafter, we will develop analyses using this unified system space. In §7, we present a monad operating over this space and provide concrete types for U, V, W, and Y to instantiate the analysis. (We cannot proceed to that discussion before discussing caching in §6.)

5.2 Evaluation

Within our formalization, we assume a configuration-centric evaluation function

$$\mathit{exec}_{\mathit{eval}}^{m,\mathit{Config},\mathit{Val}} : (\mathit{Config} \to m(\mathit{Val} \times V)) \to \mathit{Config} \to m(\mathit{Val} \times V)$$

which (as suggested by the type) is open-recursive, so that all apparently recursive calls are intercepted by the functional argument, and which is parameterized by a monad $m : \mathit{Type} \to \mathit{Type}$, a configuration set Config, and a value set Val. We demonstrate in §7 how to obtain such a function from eval^m.

Monotonicity condition In this work, we rely on the assumption that, for a fixed m, Config, and Val, $\mathit{exec}_{\mathit{eval}}^{m,\mathit{Config},\mathit{Val}}$ is monotonic with respect to its parameters, provided its functional argument is monotonic with respect to its parameters.

5.3 Program Meaning

A system space definition does not itself enforce a particular sensitivity on any given component but is only reflective of it. In this section, we define the meaning of these sensitivities by formally defining when an element of the system space exhibits them.

These sensitivities are defined relative to a program evaluation that treats all components path-sensitively. To support this, we define the set of configurations Config_{ps} and the set of results $\mathit{Result}_{ps}^{\mathit{Val}}$, each of which includes all components.

$$\mathit{Config}_{ps} := \mathit{Control} \times V \times W \times Y \qquad \mathit{Result}_{ps}^{\mathit{Val}} := \mathcal{P}(\mathit{Val} \times V \times W \times Y)$$

We instantiate $\mathit{exec}_{\mathit{eval}}^{m,\mathit{Config},\mathit{Val}}$ with a type constructor PS^{Val} (for *path-sensitive*) defined

$$PS^{\mathit{Val}} : \mathit{Type} \to \mathit{Type}$$
$$PS^{\mathit{Val}}(a) := \mathcal{P}(a \times W \times Y) \times \mathcal{P}(\mathit{Config}_{ps}) \times [\mathit{Config}_{ps} \mapsto \mathit{Result}_{ps}^{\mathit{Val}}]$$

so that

$$PS^{\mathit{Val}}(\mathit{Val} \times V) = \mathit{Result}_{ps}^{\mathit{Val}} \times \mathcal{P}(\mathit{Config}_{ps}) \times [\mathit{Config}_{ps} \mapsto \mathit{Result}_{ps}^{\mathit{Val}}].$$

To reduce notational overhead, we will henceforth omit the Val superscript, but PS, Result_{ps}, and $\mathit{exec}_{\mathit{eval}}^{m,\mathit{Config}}$ remain parameterized by a Val domain.

The motivation for this definition of PS is to use the open-recursive feature of $\mathit{exec}_{\mathit{eval}}^{PS}$ (i.e. $\mathit{exec}_{\mathit{eval}}^{m,\mathit{Config},\mathit{Val}}$ instantiated with PS and Config_{ps}) to instrument evaluation to record reached configurations and returned results within the set and cache components introduced by PS. To achieve this, we first define $\mathit{exec}_{\mathit{cache}}^{PS}$, which intercepts an execution and records the encountered configuration and its results.

$$\mathit{exec}_{\mathit{cache}}^{PS} : (\mathit{Config}_{ps} \to PS(\mathit{Val} \times V)) \to \mathit{Config}_{ps} \to PS(\mathit{Val} \times B)$$
$$\mathit{exec}_{\mathit{cache}}^{PS} := \lambda \mathit{exec}.\lambda\varsigma.(\emptyset, \{\varsigma\}, [\varsigma \mapsto r]) \sqcup (r, R, \$) \text{ where } (r, R, \$) := \mathit{exec}(\varsigma)$$

This function evaluates a given configuration ς using its functional argument *exec* and intercepts the resultant triple containing the evaluation result r, set of reachable configurations R, and cache \$. It joins this triple with one in which ς is resident of the reachable configuration set and ς is associated with r in the cache. The join operation distributes componentwise as is defined as set union for the result and reachable configuration sets.

We compose $exec^{PS}_{cache}$ with $exec^{PS}_{eval}$ to define $exec^{PS}_{path}$, which evaluates a configuration and caches its result, as follows.

$$exec^{PS}_{path} : (Config_{ps} \to PS(Val \times V)) \to Config_{ps} \to PS(Val \times V)$$
$$exec^{PS}_{path} := \lambda exec.exec^{PS}_{cache}(exec^{PS}_{eval}(exec))$$

Using $exec^{PS}_{path}$, we define the meaning $[\![pr]\!]_{eval}$ of a program pr with respect to the evaluation function $exec^{PS}_{eval}$.

Definition 1 (Program Meaning). *The* meaning $[\![pr]\!]_{eval}$ *of a program pr with respect to an evaluation function* $exec_{eval}$ *is given by the following.*

$$[\![pr]\!]_{eval} : \mathcal{P}(Config_{ps}) \times [Config_{ps} \mapsto Result_{ps}]$$
$$[\![pr]\!]_{eval} := \bigsqcup_{n \geq 0} \pi_{2\times 3}((exec^{PS}_{path})^n(\bot_{exec})(\mathcal{I}_{ps}(pr)))$$

where $\mathcal{I}_{ps}(pr) := (pr, u_0, v_0, \bot_W, \bot_Y)$; $\bot_{exec}$ is the function $\lambda\varsigma.(\emptyset, \emptyset, \bot)$ producing the empty result, empty reachability set, and empty cache; and $\pi_{2\times 3}$ projects the second and third components from a triple. Thus, a program meaning with respect to an evaluation function is a pair of a set of reachable configurations and a map from configurations to results.

5.4 Analysis Validity

We now define the validity of an analysis with respect to a program meaning. Per §5, an analysis is an element $(R, \$^w, \$, y) : \Sigma$ of the unified system space where

- R is a set of reachable configurations (which, in contrast to configurations in a program meaning, do not contain path-insensitive components),
- $\$^w$ maps configurations to the flow-sensitive quantity at evaluation start,
- \$ maps configurations to a pair of their results and the flow-sensitive quantity at evaluation end, and
- y is the flow-insensitive quantity.

Definition 2 (Analysis Validity). *An analysis* $(R, \$^w, \$, y)$ *is* valid *for a program pr, written* $(R, \$^w, \$, y) \vDash_{eval} pr$*, if, for* $[\![pr]\!]_{eval} = (R_{ps}, \$_{ps})$*, for each* $\varsigma \in R_{ps}$*, where* $\varsigma = (e, u_0, v_0, w_0, y_0)$ *and letting* $\lfloor\varsigma\rfloor = (e, u_0, v_0)$*, and for each* $(v, v_1, w_1, y_1) \in \$_{ps}(\varsigma)$*,*

1. $\lfloor\varsigma\rfloor \in R$,

2. $w_0 \sqsubseteq_W \$^w(\lfloor \varsigma \rfloor)$,
3. $(\{(v, v_1)\}, w_1) \sqsubseteq_{\mathcal{P}(Val \times V) \times W} \$(\lfloor \varsigma \rfloor)$,
4. $y_0 \sqsubseteq_Y y$, *and*
5. $y_1 \sqsubseteq_Y y$.

Condition 1 ensures that the analysis considers the configuration reachable. Condition 2 ensures that the flow-sensitive component at entry is included; condition 3 ensures that each result and flow-sensitive component at exit is included. Conditions 4 and 5 ensure that the flow-insensitive quantity is included as manifest at entry and exit, respectively.

6 A Caching Algorithm

In this section, we present a caching algorithm which computes a valid analysis using standard Kleene iteration.

By design, this algorithm is not clever (and is correspondingly inefficient). Each iteration, it visits the configurations encountered in the previous iteration. At the visit of each configuration, it records its evaluation behavior, including the configurations it encounters and the results it produces. At the outset, only the initial program configuration is considered reachable.

First, we define the sets of configurations $Config_{full}$ and results $Result_{full}$ used by the analysis.

$$Config_{full} := Control \times V \qquad Result_{full} := \mathcal{P}(Val \times V)$$

We then instantiate $exec_{eval}^{m,Config}$ with the type constructor $Full_{naive}$ defined

$$\begin{aligned} &Full_{naive} : Type \to Type \\ &Full_{naive}(a) := W \to Y \to (Config_{full} \mapsto Result_{full} \times W) \\ &\qquad\qquad \to \mathcal{P}(a) \times W \times Y \times \mathcal{P}(Config_{full}) \times (Config_{full} \mapsto W). \end{aligned}$$

We then define a function $exec_{stub}^{Full}$ as

$$\begin{aligned} &exec_{stub}^{Full} : Config_{full} \to Full_{naive}(Val \times V) \\ &exec_{stub}^{Full}(\varsigma)(w)(y)(\$) := (\$(\varsigma), \bot_Y, \{\varsigma\}, [\varsigma \mapsto w]). \end{aligned}$$

The function $exec_{stub}^{Full}$ is *not* defined in an open-recursive style, so we can pass it to $exec_{eval}^{Full}$ to produce an execution function as follows.

$$\begin{aligned} &exec^{Full} : Config_{full} \to Full_{naive}(Val \times V) \\ &exec^{Full} := exec_{eval}^{Full}(exec_{stub}^{Full}) \end{aligned}$$

Under these definitions, when $exec^{Full}$ makes an (apparently) recursive call, the configuration is recorded as reachable, its result is pulled directly from the cache, and the flow-sensitive quantity is associated with the configuration. The flow-insensitive component Y is included in the output so that $exec_{eval}^{Full}$ has the opportunity to register a contribution during evaluation (such as writing to the store).

6.1 An Analysis Transfer Function

With $exec^{Full}$ in hand, we define the transfer function G which uses it to evaluate each reachable configuration in the context of the provided flow-sensitive and result cache and flow-insensitive quantity as follows (note that the set comprehension is broken over two lines)

$$
\begin{aligned}
&G : \Sigma \to \Sigma \\
&G(R_0, \$_0^w, \$_0, y_0) := \bigsqcup \{(R, \$^w, [\varsigma \mapsto r], y) : \\
&\qquad\qquad \varsigma \in R_0, (r, d, R, \$^w) := exec^{Full}(\varsigma)(\$_0^w(\varsigma))(y_0)(\$_0)\}.
\end{aligned}
$$

Because the transfer function G does not seed the reachable set with the initial configuration, it alone cannot be iterated to compute the fixed point. To rectify this, we define the full transfer function F in terms of G and which ensures that the initial configuration is considered reachable as follows

$$
\begin{aligned}
&F : \Sigma \to \Sigma \\
&F(R_0, \$_0^c, \$_0, d_0) := G(R_0, \$_0^w, \$_0, y_0) \sqcup \\
&\qquad\qquad (\{\mathcal{I}_{full}(pr)\}, \bot_{Config_{full} \mapsto W}, \bot_{Config_{full} \mapsto Result_{full} \times W}, \bot_Y)
\end{aligned}
$$

where $\mathcal{I}_{full}(pr) := (pr, u_0, v_0)$.

6.2 *F* Yields a Valid Analysis

We now show that a fixed point of F is a valid analysis. First, we establish a few lemmas.

Lemma 1. *Σ has no infinite ascending chains.*

It follows immediately from the fact that no component of Σ has any infinite ascending chains, which in each case holds by assumption or inspection.

Lemma 2. *G is monotonic in R, $\$^w$, $\$$, and y.*

G is monotonic in each component by definition and by the monotonicity condition of the evaluation function in §5.2.

Lemma 3. *This caching algorithm is computable.*

We obtain this from the Kleene fixed point theorem as follows: F is computable so each iteration to compute the ascending chain is computable. F is monotonic and the unified state space Σ is a directed-complete partial order, so F has a least fixed point which Kleene iteration yields.

Theorem 1. *If $(R, \$^w, \$, y)$ is a fixed point of F for program pr with respect to some $exec_{eval}^{Full}$, then $(R, \$^w, \$, y) \vDash_{eval} pr$.*

The proof is similar to the proof of Darais [7]. One difference is that Darais defines their semantics as a big step relation whereas we have defined our semantics as a function. However, the key behaviors of reachability and evaluation are consistent across both semantics. Accordingly, the proof itself proceeds by mutual induction of reachability and evaluation over the call trace.

7 Completely Instantiating the Abstract Semantics

In §3, we parameterized the definitional interpreter with an abstract value domain and timestamp set. Armed with a framework of full path and flow sensitivity, we are ready to complete its parameterization with a monad instance. We define this instance in two layers: an instance for the caching algorithm type transformer $\mathit{Full}_{\mathit{naive}}$ of §6—which we abbreviate to *Full* that insulates access to U and V and an instance on top of it which exposes access to U and V.

Once this instance is defined, we instantiate the control-flow sensitive parameters to obtain an actual analyzer.

7.1 Caching Monad Instance

We first define a monad instance for *Full* from §6 with state and nondetermism effects in Figure 8. To $\mathit{return}^{\mathit{Full}}$ a value injects it into a singleton set, returns

$$
\begin{aligned}
&\mathit{return}^{\mathit{Full}} : \forall A.A \to \mathit{Full}(A)\\
&\mathit{return}^{\mathit{Full}}(x)(w)(y)(\$) := (\{(x)\}, w, y, \emptyset, \bot)\\
&\mathit{bind}^{\mathit{Full}} : \forall A.\forall B.\mathit{Full}(A) \to (A \to \mathit{Full}(B)) \to \mathit{Full}(B)\\
&\mathit{bind}^{\mathit{Full}}(\mathit{cmp})(f)(w)(y) := (\emptyset, \bot_W, \bot_Y, R, \$) \sqcup f(x_1)(w')(y') \sqcup \cdots \sqcup f(x_n)(w')(y')\\
&\qquad\qquad\qquad\qquad \text{where } (\{x_1, \ldots, x_n\}, w', y', R, \$) := \mathit{cmp}(w)(y)\\
&\_\langle + \rangle^{\mathit{Full}}\_ : \forall A.\mathit{Full}(A) \times \mathit{Full}(A) \to \mathit{Full}(A)\\
&\mathit{cmp}_0 \langle + \rangle^{\mathit{Full}} \mathit{cmp}_1 := \lambda w.\lambda y.\mathit{cmp}_0(w)(y) \sqcup \mathit{cmp}_1(w)(y)\\
&\mathit{mzero}^{\mathit{Full}} : \forall A.\mathit{Full}(A)\\
&\mathit{mzero}^{\mathit{Full}}(w_0)(y_0) := (\emptyset, \bot_W, \bot_Y, \emptyset, \bot)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{get}_W^{\mathit{Full}} : \mathit{Full}(W) && \mathit{get}_Y^{\mathit{Full}} : \mathit{Full}(Y)\\
&\mathit{get}_W^{\mathit{Full}}(w)(y) := \mathit{return}^{\mathit{Full}}(w)(w)(y) && \mathit{get}_Y^{\mathit{Full}}(w)(y) := \mathit{return}^{\mathit{Full}}(y)(w)(y)\\
&\mathit{put}_W^{\mathit{Full}} : W \to \mathit{Full}(1) && \mathit{put}_Y^{\mathit{Full}} : Y \to \mathit{Full}(1)\\
&\mathit{put}_W^{\mathit{Full}}(w)(\_)(y) := \mathit{return}^{\mathit{Full}}(\langle\rangle)(w)(y) && \mathit{put}_Y^{\mathit{Full}}(y)(w)(\_) := \mathit{return}^{\mathit{Full}}(\langle\rangle)(w)(y)
\end{aligned}
$$

Fig. 8. A monad instance for *Full* with state and nondeterminism effects

the provided state w and y, and produces an empty reachability set and flow-sensitive cache. To $\mathit{bind}^{\mathit{Full}}$ a computation to a function runs the computation and feeds each result to the function, joining the results. The reachability set and flow-sensitive cache of the computation's execution are included in the final result. The nondeterministic choice between two computations runs both and joins their results; a failing computation returns no results. The state effects for W and Y are standard.

$$\gamma_{Full_{eval}} : (\mathbf{Exp} \to Full_{eval}(\widehat{Val})) \to \widehat{Config} \to Full(\widehat{Val} \times V)$$
$$\gamma_{Full_{eval}}(eval)(e, u, v) := eval(e)(u)(v)$$
$$\alpha_{Full_{eval}} : (\widehat{Config} \to Full(\widehat{Val} \times V)) \to \mathbf{Exp} \to Full_{eval}(\widehat{Val})$$
$$\alpha_{Full_{eval}}(exec)(e)(u)(v) := exec(e, u, v)$$

Fig. 9. The $\gamma_{Full_{eval}}/\alpha_{Full_{eval}}$ conversion pair for the caching and evaluation monads

Lemma 4. *The instance for Full satisfies the monad, nondeterminism, and state laws.*

Adherence to the laws follows from the definitions.

7.2 Evaluation Monad Instance

We now define $Full_{eval}$, a type transformer in terms of $Full$.

$$Full_{eval} : Type \to Type$$
$$Full_{eval}(a) := U \to V \to Full(a \times V)$$

$Full_{eval}$ admits a straightforward monad instance, which we define in an accompanying technical report [8]. Whereas $Full$ incorporates the components U and V in a way oblivious to the underlying caching, $Full_{eval}$ exposes them. This two-layered approach insulates the caching from evaluation, allowing us to change the caching strategy without disturbing this evaluation definition. In Figure 9, we define the $\gamma_{Full_{eval}}/\alpha_{Full_{eval}}$ pair to convert between these two monads.

7.3 An Instantiated Analyzer

We now put all of the pieces together to produce a concretely-instantiated analyzer. We define configurations and results generically as follows.

$$\widehat{Config} := Control \times V \qquad \widehat{Result} := \mathcal{P}(\widehat{Val} \times V)$$

The final requirement for the caching algorithm is an evaluation function from configurations to results. We use $\gamma_{Full_{eval}}$ and $\alpha_{Full_{eval}}$ to wrap the evaluation function instantiated with $Full_{eval}$ as follows.

$$exec^{Full} : (\widehat{Config} \to Full(\widehat{Val} \times V)) \to \widehat{Config} \to Full(\widehat{Val} \times V)$$
$$exec^{Full}(exec) := \gamma_{Full_{eval}}(eval^{Full_{eval}}(\alpha_{Full_{eval}}(exec)))$$

This is of precisely the type required by the caching algorithm and we can now select the control-flow sensitivity of analysis components by how we instantiate U, V, W, and Y. The generic evaluation function $eval^m$ expects a per-path environment and timestamp, which dictates that the per-path component U must be instantiated with $\widehat{Env} \times \widehat{Time}$ but we can select the sensitivity of the store.

Path-sensitive analysis We select path sensitivity for the store by instantiating V as $\widehat{Store}$ and W and Y as the unit domain.

$$U = \widehat{Env} \times \widehat{Time} \qquad V = \widehat{Store} \qquad W = 1 \qquad Y = 1$$

What remains is to define the accessors of configuration components, such as environments and stores, in terms of the monadic effects, which we provide in an accompanying technical report [8].

Flow-sensitive analysis We select flow sensitivity for the store by instead instantiating W as $\widehat{Store}$ and V and Y as the unit domain.

$$U = \widehat{Env} \times \widehat{Time} \qquad V = 1 \qquad W = \widehat{Store} \qquad Y = 1$$

An accompanying technical report [8] contains an analysis treating the store flow-sensitively using the caching algorithm presented in §6.

7.4 Semantics Independence

As the instantiation in this section demonstrates, our framework requires of the semantics only (1) a functional interface with an open-recursive type (2) that is monotonic in each of its arguments. Beyond these requirements, the analyzer makes no assumptions about the analyzed language, and in that sense the framework offers *semantics independence*.

As currently formulated, this function may require the analyzer to instantiate the timestamp—which offers context sensitivity—as a per-path or path-sensitive quantity. If our framework were extended to include context sensitivity in the purview of control-flow sensitivity, as Kim *et al.* [17] do in their framework, the evaluation function would place effectively no constraints on the framework. (We discuss this work further in §10.)

8 Accounting for *Abstracting Definitional Interpreters*

We have used monads extensively to demonstrate our framework in action, an approach inherited from the work on modular abstract interpreters on which it is based [6]. In fact, Darais *et al.* [6] systematize the construction of these monads using monad transformers [18] which correspond to analogous "system space transformers" called *Galois transformers*.

Galois transformers (GT) account for the relationship between program points and analysis components—and therefore the control-flow sensitivity with which the analysis treats them—through analysis effects. To wit, the path-sensitive system space $\mathcal{P}(Point \times B)$ is understood as the application of a state transformer $\cdot \times B$ followed by the application of a nondeterminism transformer $\mathcal{P}(\cdot)$. The flow-insensitive system space $\mathcal{P}(Point) \times B$ is understood as the application of the same transformers but in the opposite order. The path-insensitive/flow-sensitive

system space $Point \mapsto B$ is produced by a single "flow-sensitive" transformer $\cdot \mapsto B$. The GT framework is designed so that the resulting analysis actually exhibits the sensitivity suggested by the induced system space and that the other aspects of its definition remain independent. The result is that one can alter the control flow sensitivity merely by swapping out a stack of system space transformers.

Later, Darais *et al.* [5] combined the systematic approach to analysis *derivation* pioneered by the *Abstracting Abstract Machines* (AAM) framework [24] and the systematic approach to analysis *construction* introduced by the GT framework into the *Abstracting Definitional Interpreters* (ADI) framework for static analyzers for higher-order languages based on definitional interpreters. However, definitional interpreters naturally lead to summary-based analyses whose state spaces differ from program point-based analyses. Consequently, analyses in the ADI framework are constructed using monad transformers just as they are in the GT framework, but they do not offer the same sensitivities that the GT framework does.

For example, we can apply state and nondeterminism transformers to a base system space of $Control \mapsto Val$. Applying them in one order yields $Control \times B \mapsto \mathcal{P}(Val \times B)$, a path-sensitive state space, whereas applying them in the other yields $Control \times B \mapsto \mathcal{P}(Val) \times B$, a state space which allows arbitrary relations between fact bases and configurations but shares a fact base among the results, which we will term *result widening*.

Result widening is not flow-insensitive, as the GT framework delivers in its native setting, nor is it flow-sensitive, because of the increased associativity between configurations and fact bases. We can situate result widening in a hierarchy of control-flow sensitivities seen in Figure 10. Result widening is not as

$Control \times B \mapsto \mathcal{P}(Val \times B)$	path sensitivity
$Control \times B \mapsto \mathcal{P}(Val) \times B$	result widening
$[Control \mapsto B] \times [Control \mapsto \mathcal{P}(Val) \times B]$	flow sensitivity
$[Control \mapsto \mathcal{P}(Val)] \times B$	flow insensitivity

Fig. 10. A hierarchy of control-flow sensitivities including result widening

precise as full path sensitivity but not as imprecise as full flow sensitivity (and path insensitivity), since the fact base remains a distinguishing component of configurations.

We can account for result widening in our framework by adding the component Z to our unified system space as follows. Letting $Con := Control \times V$, we define

$$\Sigma_{rw} := \mathcal{P}(Con \times Z) \times [Con \times Z \mapsto W] \times [Con \times Z \mapsto \mathcal{P}(Val \times V) \times W \times Z] \times Y$$

and provide analyses access to Z through yet another state effect.

9 Implementation

We have implemented the framework with three caching algorithms: the naïve caching algorithm presented in § 6, the coinductive caching algorithm of ADI [5], and a caching algorithm based on a dependency-tracking fixed point solver cast as a monad [25]. We verified that each algorithm produced the same results.

We do not set out here to empirically establish the performance characteristics of the framework but to instead demonstrate that

1. the framework is readily deployed within an implementation,
2. the framework's conceptualization of control-flow sensitivity is robust enough to be implemented under a variety of different caching algorithms, and
3. the size of the state spaces within a big-step conceptualization of the different sensitivities matches intuitions developed in small-step settings.

Our analysis is a 0CFA with abstract garbage collection [20]. To implement abstract garbage collection in a definitional interpreter setting, we use the technique of Darais *et al.* [5] wherein each configuration has a per-path component which includes the set of stack addresses.

Although the presentation of our framework uses monads, we implemented our framework in a language with no explicit support for monads. In fact, our implementation, closely following the presentation here, uses no more than higher-order functions.

For each caching algorithm and control-flow sensitivity, we deployed the analysis on seven programs. Six of the programs are small programs with patterns that yield different behaviors under different sensitivities. The seventh program is a SAT solver and is the most substantial program on which we deployed the analysis. We timed 100 iterations of the analysis on each program; Figure 11 plots and analyzes the results.

10 Related Work

The *Galois Transformers* framework of Darais *et al.* [6] provides the means to construct an analyzer using monad transformers to get language-independent flow sensitivity properties and soundness theorems by construction. The *Abstracting Definitional Interpreters* framework [5] used monad transformers in the same way, obtaining soundness by construction, but exhibiting a limited range of flow sensitivity with that mechanism. This work offers a framework which restores the full range of flow sensitivity in the setting of definitional interpreters, offering those sensitivities in concert with the higher control precision definitional interpreters offer. It does not hold the modular construction via monad transformers as essential, though it is easy to see where monad transformers would aid the construction of the monads. However, where Galois transformers

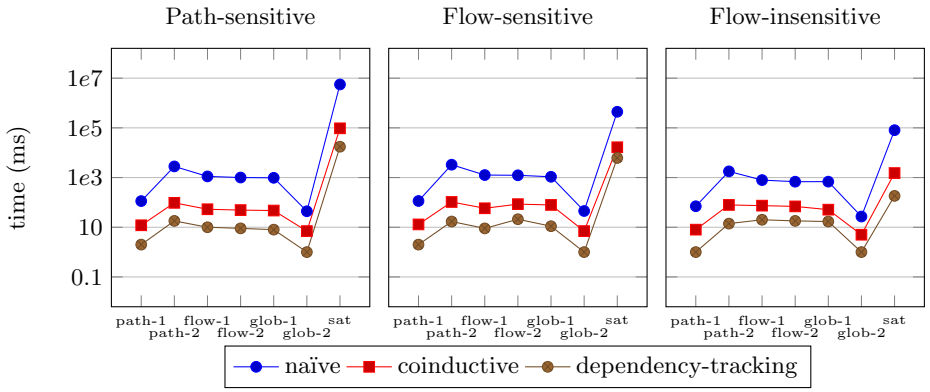

Fig. 11. This plot presents benchmark results for seven programs at three sensitivites each computed by three caching algorithms. The y-axis is log-scale. We can see two trends within these results. First, for a given control-flow sensitivity, the analysis time depends on the caching algorithm used, with the naïve algorithm taking the most time and a dependency-tracking algorithm taking the least. Second, the analysis time decreases as the store moves from path sensitivity to insensitivity and again as it moves from flow sensitivity to flow insensitivity. These are the same trends that arise across algorithms and control-flow sensitivities in the small-step setting.

allow additional components at a particular sensitivity to be added, analyzers in this framework manage exactly one component at each sensitivity, which requires the entire analysis instantiation to cooperate.

Hardekopf *et al.* [15] also offer a means to separate the flow sensitivity of the analysis from other aspects of its specification by casting it as a widening operator. Like Darais *et al.* [6], they present their work in a setting where continuations are reified within the semantics and argue that abstracting control requires such a handle on the control. They further argue that semantics which do not offer this handle, such as big-step semantics, therefore offer "no way to abstract and over-approximate control flow". In this work, we have argued that achieving different sensitivities in this setting is possible, and provided an exhibition of flow sensitivity tailored to the summary-oriented nature of the semantics.

Kim *et al.* [17] describe a generic sensitivity framework which expresses the natural disjunctive reading of a sensitive analysis as a conjunctive reading where each conjunct is predicated by preconditions which must hold for the property to obtain. This alternative reading allows many properties to be treated independently while avoiding a Cartesian explosion of states that a disjunction reading produces. Our framework provides a similar reading where each configuration acts as a precondition for a flow-sensitive quanitity or result to obtain. This framework essentially expresses value sensitivity which subsumes path sensitivity by predicating properties on the boolean values selecting each branch. Our framework provides fact-base associativity which manifests path sensitivity in a similar way—admitting distinctions between particular program quantities—but

work is needed to formally compare the expressive power of each. This framework is state-based, providing a description of the reachable states in terms of a conjunction of implications. In contrast, our framework is summary-based, providing a description of evaluations in terms of a disjunction of configurations and their results. Our framework is also able to handle higher-order behavior, a capability not discussed in this work.

Handjieva and Tzolovski [12] present a technique to increase the precision of a static analysis by lifting the abstract domain to finite sets of abstract properties labelled by some residue of control flow. The soundness of this work relies on the ability to succinctly characterize subgraphs of the control-flow graph by their topology, which is demonstrated for explicit, static control-flow graphs. Our framework operates without knowledge of the control flow graph in which control-flow actions are signalled by returns and inner recursive calls of a semantic function. As with Kim *et al.*, this work is presented over a first-order language for a point-based analysis.

Bourdoncle [1] presents a technique to derive more-sophisticated abstract interpretations from simpler ones that are responsive to the program, in part by associating control points and abstract values within the analysis. This approach leads to a similar state space to a path-sensitive analysis in our framework. Moreover, this work allows for broad notion of *control point* which, translated to our framework, includes functions of the stack, environment, and timestamp.

11 Discussion and Future Work

The framework we have presented computes an account of program behavior that (1) requires only a functional interface to the language semantics, (2) allows the control-flow sensitivity of different analysis elements to be adjusted independent of this semantics, and (3) is in terms of summaries of evaluation.

Although this framework is heavily inspired by Galois transformers, it offers only a semantics-independent mechanism to adjust control-flow sensitivity in a summary-based setting, and not a genuine analysis transformer which introduces an analysis component to be treated with a particular sensitivity. Future work will develop genuine Galois transformers for the definitional interpreter setting.

Unlike other frameworks [17, 15], our framework instantiates the notion and degree of context sensitivity up front, and requires the cooperation of the semantics. Future work will address incorporating context sensitivity as a parameter of the framework that disturbs neither the functional interface to the semantics nor summary-based result of analysis.

Finally, the functional interface to the semantics provides very coarse insulation from semantic details to other aspects of the analysis, but this coarseness limits the degree to which the analysis can be tuned to a particular semantics. Future work is to explore the tailoring facilities of different semantic interfaces.

References

1. Bourdoncle, F.: Abstract interpretation by dynamic partitioning. J. Funct. Program. **2**(4), 407–423 (1992). https://doi.org/10.1017/S0956796800000496, https://doi.org/10.1017/S0956796800000496
2. Chase, D.R., Wegman, M.N., Zadeck, F.K.: Analysis of pointers and structures. In: Fischer, B.N. (ed.) Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990. pp. 296–310. ACM (1990). https://doi.org/10.1145/93542.93585, https://doi.org/10.1145/93542.93585
3. Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 159–169. POPL '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1328438.1328460, https://doi.org/10.1145/1328438.1328460
4. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 232–245. POPL '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/158511.158639, https://doi.org/10.1145/158511.158639
5. Darais, D., Labich, N., Nguyen, P.C., Van Horn, D.: Abstracting definitional interpreters (functional pearl). Proceedings of the ACM on Programming Languages **1**(ICFP), 12:1–12:25 (Aug 2017). https://doi.org/10.1145/3110256
6. Darais, D., Might, M., Van Horn, D.: Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 552–571. OOPSLA 2015, ACM, New York, NY, USA (Oct 2015). https://doi.org/10.1145/2814270.2814308
7. Darais, D.C.: Mechanizing Abstract Interpretation. Ph.D. thesis, University of Maryland, College Park, MD, USA (2017). https://doi.org/10.13016/M2J96097D, https://hdl.handle.net/1903/19989
8. Germane, K.: Full control-flow sensitivity for definitional interpreters. Tech. rep., Brigham Young University (2024)
9. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 2–14. ACM (2011). https://doi.org/10.1145/2034773.2034777, https://doi.org/10.1145/2034773.2034777
10. Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 407–420. ACM (2016). https://doi.org/10.1145/2951913.2951936, https://doi.org/10.1145/2951913.2951936
11. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Sec. **8**(6), 399–422 (2009). https://doi.org/10.1007/S10207-009-0086-1, https://doi.org/10.1007/s10207-009-0086-1

12. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1503, pp. 200–214. Springer (1998). https://doi.org/10.1007/3-540-49727-7"˙12, https://doi.org/10.1007/3-540-49727-7_12
13. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009. pp. 226–238. ACM (2009). https://doi.org/10.1145/1480881.1480911
14. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011. pp. 289–298. IEEE Computer Society (2011). https://doi.org/10.1109/CGO.2011.5764696, https://doi.org/10.1109/CGO.2011.5764696
15. Hardekopf, B., Wiedermann, B., Churchill, B.R., Kashyap, V.: Widening for control-flow. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8318, pp. 472–491. Springer (2014). https://doi.org/10.1007/978-3-642-54013-4"˙26, https://doi.org/10.1007/978-3-642-54013-4_26
16. Hudak, P.: A semantic model of reference counting and its abstraction (detailed summary). In: Scherlis, W.L., Williams, J.H., Gabriel, R.P. (eds.) Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986. pp. 351–363. ACM (1986). https://doi.org/10.1145/319838.319876, https://doi.org/10.1145/319838.319876
17. Kim, S., Rival, X., Ryu, S.: A theoretical foundation of sensitivity in an abstract interpretation framework. ACM Trans. Program. Lang. Syst. **40**(3), 13:1–13:44 (2018). https://doi.org/10.1145/3230624, https://doi.org/10.1145/3230624
18. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 333–343. POPL '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199528, https://doi.org/10.1145/199448.199528
19. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, USA (2007), https://hdl.handle.net/1853/16289
20. Might, M., Shivers, O.: Improving flow analyses via gammaCFA: abstract garbage collection and counting. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming. pp. 13–25. ICFP '06, ACM, New York, NY, USA (Sep 2006). https://doi.org/10.1145/1159803.1159807
21. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 305–315. PLDI '10, ACM, New York, NY, USA (Jun 2010). https://doi.org/10.1145/1806596.1806631
22. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
23. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the 38th Annual

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 17–30. POPL '11, ACM, New York, NY, USA (Jan 2011). https://doi.org/10.1145/1926385.1926390

24. Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM, New York, NY, USA (Sep 2010). https://doi.org/10.1145/1863543.1863553
25. Vandenbroucke, A., Schrijvers, T., Piessens, F.: Fixing non-determinism. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. IFL '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2897336.2897342, https://doi.org/10.1145/2897336.2897342
26. Wei, G., Decker, J., Rompf, T.: Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). Proceedings of the ACM on Programming Languages **2**(ICFP), 105:1–105:28 (Jul 2018). https://doi.org/10.1145/3236800