Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis

Abstract

The design and implementation of static analyzers have becoming increasingly systematic. In fact, for large classes of analyzers, design and implementation have remained seemingly (and now stubbornly) on the verge of full mechanization for several years. A stumbling block in full mechanization has been the *ad hoc* nature of soundness proofs accompanying each analyzer. While design and implementation is largely systematic, soundness proofs can change significantly with what seem like minor changes to the semantics and analyzers themselves. An achievement of this work is to systematize, parmaterize and modularize the proofs of soundness, so as to make them composoble across analytic properties.

We solve the problem of systematically constructing static analyzers by introducing the notion of a *Galois transformer*, a monad transformer that satisfies the properties of a Galois connection. In concert with a monadic interpreter, we are able to define a library of monad transformers that implement and represent classic context-, path-, and heap-(in-?)sensitive building blocks. Moreover, these can be composed together to realize a program analysis monad, *independent of the language being analyzed*.

Significantly, a Galois transformer can be proved sound once and for all, making it a reusable analysis component. As new analysis features and abstractions are developed and mixed in, soundness proofs need not be reconstructed, as the composition of a monad transfomer stack is sound by virtue of its constituents. Galois transformers provide a viable foundation for resuable and composable metatheory for program analysis. Finally, these Galois transformers shift the level of abstraction in analysis design and implementation to a level where non-specialists have the ability to synthesize sound analyzers over a number of parameters.

1. Introduction

Traditional practice in the program analysis literature, be it for points-to, flow, shape analysis or others, is to fix a language and its abstraction (a computable, sound approximation to the "concrete" semantics of the language) and investigate its effectiveness. These one-off abstractions require effort to design and prove sound. Consequently later work has focused on endowing the abstraction with a number of knobs, levers, and dials to tune precision and compute efficiently. These parameters come in various forms with overloaded meanings such as object, context, path, and heap sensitivities, or some combination thereof. These efforts develop families of analyses for a specific language and prove the framework sound.

But even this framework approach suffers from many of the same drawbacks as the one-off analyzers. They are language specific, preventing reuse across languages and thus requiring similar abstraction implementations and soundness proofs. This process is difficult and error prone. It results in a cottage industry of research papers on varying frameworks for varying languages. It prevents fruitful insights and results developed in one paradigm from being applied to others.

In this paper, we propose an alternative approach to structuring and implementing program analysis. Inspired by **?**'s *Monad transformers for modular interpreters* [**?**], we propose to use concrete interpreters in monadic style. As we show, classical program abstractions can be embodied as language-independent monads. Moreover, these abstractions can be written as monad transformers, thereby allowing their composition to achieve new forms of analysis. We show that these monad transfomers obey the properties of *Galois connections* [Cousot and Cousot 1977] and introduce the concept of a *Galois transfomer*, a monad transfomer that forms a Galois connection.

Most significantly, these Galois transformers can be proved sound once and for all. Abstract interpreters, which take the form of monad transformer stacks coupled together with a monadic interpreter, inherit the soundness properties of each element in the stack. This approach enables reuse of abstractions across languages and lays the foundation for a modular metatheory of program analysis.

1.1 Contributions

Our contributions are:

- A compositional meta-theory framework for building correct-by-construction abstract interpreters. This frame-

$$
\begin{aligned}
i &\in \mathbb{Z} \\
x &\in \mathit{Var} \\
a &\in \mathit{Atom} ::= i \mid x \mid \underline{\boldsymbol{\lambda}}(x).e \\
\oplus &\in \mathit{IOp} \quad ::= + \mid - \\
\odot &\in \mathit{Op} \quad ::= \oplus \mid @ \\
e &\in \mathit{Exp} \quad ::= a \mid e \odot e \mid \underline{\mathbf{if0}}(e)\{e\}\{e\}
\end{aligned}
$$

Figure 1: λIF

work is built using a restricted class of monad transformers.

- An isolated understanding of flow and path-sensitivity for static analysis. We understand this spectrum as mere variations in the order of monad transformer composition in our framework.

1.2 Outline

We will demonstrate our framework by example, walking the reader through the design and implementation of an abstract interpreter. Section 2 gives the concrete semantics for a small functional language. Section 3 gives a brief tutorial on the path- and flow-sensitivity in the setting of our example language. Section 4 describes the parameters of our analysis, one of which is the interpreter monad. Section 5 shows the full definition of a highly parameterized monadic interpreter. Section 6 shows how to recover concrete and abstract interpreters. Section 7 shows how to manipulate the path- and flow-sensitivity of the interpreter through variations in the monad. Section 8 demonstrates our compositional meta-theory framework built on monad transformers. Section 9 briefly discusses our implementation of the framework in Haskell. Section 10 discusses related work and Section 11 concludes.

2. Semantics

To demonstrate our framework we design an abstract interpreter for **λIF**, a simple applied lambda calculus shown in Figure 1. **λIF** extends traditional lambda calculus with integers, addition, subtraction and conditionals. We use the operator **@** as explicit syntax for function application. This allows for *Op* to be a single syntactic class for all operators and simplifies the presentation.

Before designing an abstract interpreter we first specify a formal semantics for **λIF**. Our semantics makes allocation explicit and separates values and continuations into separate stores. Our approach to analysis will be to design a configurable interpreter that is capable of mirroring these semantics.

The state space Σ for **λIF** is a standard CESK machine augmented with a separate store for continuation values:

$$
\begin{aligned}
\tau &\in \mathit{Time} &&:= \mathbb{Z} \\
l &\in \mathit{Addr} &&:= \mathit{Var} \times \mathit{Time} \\
\rho &\in \mathit{Env} &&:= \mathit{Var} \rightharpoonup \mathit{Addr} \\
\sigma &\in \mathit{Store} &&:= \mathit{Addr} \rightharpoonup \mathit{Val} \\
c &\in \mathit{Clo} &&::= \langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle \\
v &\in \mathit{Val} &&::= i \mid c \\
\kappa l &\in \mathit{KAddr} &&:= \mathit{Time} \\
\kappa\sigma &\in \mathit{KStore} &&:= \mathit{KAddr} \rightharpoonup \mathit{Frame} \times \mathit{KAddr} \\
fr &\in \mathit{Frame} &&::= \langle \square \odot e \rangle \mid \langle v \odot \square \rangle \mid \langle \underline{\mathbf{if0}}(\square)\{e\}\{e\} \rangle \\
\varsigma &\in \Sigma &&::= \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{KAddr} \times \mathit{KStore}
\end{aligned}
$$

Atomic expressions are denoted by $A[\![\_,\_,\_]\!]$:

$$
\begin{aligned}
&A[\![\_,\_,\_]\!] \in \mathit{Env} \times \mathit{Store} \times \mathit{Atom} \rightharpoonup \mathit{Val} \\
&A[\![\rho,\sigma,i]\!] := i \\
&A[\![\rho,\sigma,x]\!] := \sigma(\rho(x)) \\
&A[\![\rho,\sigma,\underline{\boldsymbol{\lambda}}(x).e]\!] := \langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle
\end{aligned}
$$

Primitive operations are denotation denoted by $\delta[\![\_,\_,\_]\!]$:

$$
\begin{aligned}
&\delta[\![\_,\_,\_]\!] \in \mathit{IOp} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\delta[\![+,i_1,i_2]\!] := i_1 + i_2 \\
&\delta[\![-,i_1,i_2]\!] := i_1 - i_2
\end{aligned}
$$

The semantics of compound expressions are given relationally via the step relation $\_ \leadsto \_$:

$$
\begin{aligned}
&\_ \leadsto \_ \in \mathcal{P}(\Sigma \times \Sigma) \\
&\langle e_1 \odot e_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle e_1, \rho, \sigma, \tau, \kappa\sigma', \tau + 1 \rangle \\
&\quad \text{where } \kappa\sigma' := \kappa\sigma[\tau \mapsto \langle \square \odot e_2 \rangle :: \kappa l] \\
&\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle e, \rho, \sigma, \tau, \kappa\sigma', \mathit{tick}(\tau) \rangle \\
&\quad \text{where} \\
&\qquad \langle \square \odot e \rangle :: \kappa l' := \kappa\sigma(\kappa l) \\
&\qquad \kappa\sigma' := \kappa\sigma[\tau \mapsto \langle A[\![\rho,\sigma,a]\!] \odot \square \rangle :: \kappa l'] \\
&\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle e, \rho'', \sigma', \kappa l', \kappa\sigma, \tau + 1 \rangle \\
&\quad \text{where} \\
&\qquad \langle \langle \underline{\boldsymbol{\lambda}}(x).e, \rho' \rangle \ @ \ \square \rangle :: \kappa l' := \kappa\sigma(\kappa l) \\
&\qquad \sigma' := \sigma[(x,\tau) \mapsto A[\![\rho,\sigma,a]\!]] \\
&\qquad \rho'' := \rho'[x \mapsto (x,\tau)] \\
&\langle i_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle i, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1 \rangle \\
&\quad \text{where} \\
&\qquad \langle i_1 \oplus \square \rangle :: \kappa l' := \kappa\sigma(\kappa l) \\
&\qquad i := \delta[\![\oplus, i_1, i_2]\!] \\
&\langle i, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle e, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1 \rangle \\
&\quad \text{where} \\
&\qquad \langle \underline{\mathbf{if0}}(\square)\{e_1\}\{e_2\} \rangle :: \kappa l' := \kappa\sigma(\kappa l) \\
&\qquad e := e_1 \ \text{ when } \ i = 0 \\
&\qquad e := e_2 \ \text{ when } \ i \neq 0
\end{aligned}
$$

Our abstract interpreter will support abstract garbage collection [Might and Shivers 2006], the concrete analogue of which is just standard garbage collection. We include garbage collection for two reasons. First, it is one of the few techniques that results in both performance *and* precision improvements for abstract interpreters. Second, later we will show how to write a monadic garbage collector, recovering both concrete and abstract garbage collection in one fell swoop.

Garbage collection is defined with a reachability function R which computes the transitively reachable address from (ρ, e) in σ:

$$\begin{aligned}
&R[_] \in \mathit{Store} \to \mathit{Env} \times \mathit{Exp} \to \mathcal{P}(\mathit{Addr})\\
&R[\sigma](\rho, e) \coloneqq \mu(X).\\
&\qquad R_0(\rho, e) \cup X \cup \{l' \mid l' \in R\text{-}\mathit{Val}(\sigma(l))\ ;\ l \in X\}
\end{aligned}$$

We write $\mu(X).f(X)$ as the least-fixed-point of a function f. This definition uses two helper functions: R_0 for computing the initial reachable set and $R\text{-}\mathit{Val}$ for computing addresses reachable from addresses.

$$\begin{aligned}
&R_0 \in \mathit{Env} \times \mathit{Exp} \to \mathcal{P}(\mathit{Addr})\\
&R_0(\rho, e) \coloneqq \{\rho(x) \mid x \in FV(e)\}\\
&R\text{-}\mathit{Val} \in \mathit{Val} \to \mathcal{P}(\mathit{Addr})\\
&R\text{-}\mathit{Val}(i) \coloneqq \{\}\\
&R\text{-}\mathit{Val}(\langle \underline{\boldsymbol{\lambda}}(x).e, \rho\rangle) \coloneqq \{\rho(x) \mid y \in FV(\underline{\boldsymbol{\lambda}}(x).e)\}
\end{aligned}$$

where FV is the standard recursive definition for computing free variables of an expression.

Analogously, KR is the set of transitively reachable continuation addresses in $\kappa\sigma$:

$$\begin{aligned}
&KR[_] \in \mathit{KStore} \to \mathit{KAddr} \to \mathcal{P}(\mathit{KAddr})\\
&KR[\kappa\sigma](\kappa l_0) \coloneqq \mu(kl*).\{\kappa l_0\} \cup \kappa l * \cup \{\pi_2(\kappa\sigma(\kappa l)) \mid \kappa l \in kl*\}
\end{aligned}$$

Our final semantics is given via the step relation $\_ \leadsto^{gc} \_$ which nondeterministically either takes a semantic step or performs garbage collection.

$$\begin{aligned}
&_ \leadsto^{gc} _ \in \mathcal{P}(\Sigma \times \Sigma)\\
&\varsigma \leadsto^{gc} \varsigma'\\
&\qquad \text{where } \varsigma \leadsto \varsigma'\\
&\langle e, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \leadsto^{gc} \langle e, \rho, \sigma', \kappa l, \kappa\sigma', \tau\rangle\\
&\qquad \text{where}\\
&\qquad\qquad \sigma' \coloneqq \{l \mapsto \sigma(l) \mid l \in R[\sigma](\rho, e)\}\\
&\qquad\qquad \kappa\sigma' \coloneqq \{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in KR[\kappa\sigma](\kappa l)\}
\end{aligned}$$

An execution of the semantics is states as the least-fixed-point of a collecting semantics:

$$\mu(X).\{\varsigma_0\} \cup X \cup \{\varsigma' \mid \varsigma \leadsto^{gc} \varsigma'\ ;\ \varsigma \in X\}$$

The analyses we present in this paper will be proven correct by establishing a Galois connection with this concrete collecting semantics.

3. Flow Properties in Analysis

One key property of a static analysis is the way it tracks *flow*. The term "flow" is heavily overloaded in static analysis. We wish to draw a sharper distinction on what is a flow property. In this paper we identify three different types of flow in analysis:

1. Path-sensitive and flow-sensitive
2. Path-insensitive and flow-sensitive
3. Path-insensitive and flow-insensitive

Consider a simple if-statement in our example language **λIF** (extended with let-bindings) where an analysis cannot determine the value of N:

$$\begin{aligned}
&1\colon \text{let } x \coloneqq \underline{\mathbf{if0}}(N)\{1\}\{-1\}\ ;\\
&2\colon \text{let } y \coloneqq \underline{\mathbf{if0}}(N)\{1\}\{-1\}\ ;\\
&3\colon e
\end{aligned}$$

Path-Sensitive Flow-Sensitive A path- and flow-sensitive analysis will track both control and data flow precisely. At program point 2 the analysis considers separate worlds:

$$\begin{aligned}
&\{N = 0,\ x = \ \ \ 1\}\\
&\{N \neq 0,\ x = -1\}
\end{aligned}$$

At program point 3 the analysis remains precise, resulting in environments:

$$\begin{aligned}
&\{N = 0,\ x = \ \ \ 1,\ y = \ \ \ 1\}\\
&\{N \neq 0,\ x = -1,\ y = -1\}
\end{aligned}$$

Path-Insensitive Flow-Sensitive A path-insensitive flow-sensitive analysis will track control flow precisely but merge the heap after control flow branches. At program point 2 the analysis considers separate worlds:

$$\begin{aligned}
&\{N = \mathit{ANY},\ x = \ \ \ 1\}\\
&\{N = \mathit{ANY},\ x = -1\}
\end{aligned}$$

At program point 3 the analysis is forced to again consider both branches, resulting in environments:

$$\begin{aligned}
&\{N = \mathit{ANY},\ x = \ \ \ 1,\ y = \ \ \ 1\}\\
&\{N = \mathit{ANY},\ x = \ \ \ 1,\ y = -1\}\\
&\{N = \mathit{ANY},\ x = -1,\ y = \ \ \ 1\}\\
&\{N = \mathit{ANY},\ x = -1,\ y = -1\}
\end{aligned}$$

Path-Insensitive Flow-Insensitive A path-insensitive flow-insensitive analysis will compute a single global set of facts that must be true at all points of execution. At program points 2 and 3 the analysis considers a single world with environment:

$$\{N = \mathit{ANY},\ \ x = \{-1, 1\}\}$$

and

$$\{N = \mathit{ANY},\ \ x = \{-1, 1\},\ \ y = \{-1, 1\}\}$$

respectively.

In our framework we capture both path- and flow-sensitivity as orthogonal parameters to our interpreter. Path-sensitivity will arise from the order of monad transformers used to construct the analysis. Flow-sensitivity will arise from the Galois connection used to map interpreters to state space transition systems. For brevity, and lack of better terms, we will abbreviate these analyses as "path-sensitive", "flow-sensitive" and "flow-insensitive". This is only ambiguous for "flow-sensitive", as path-sensitivity implies flow-sensitivity, and flow-insensitivity implies path-insensitivity.

4. Analysis Parameters

Before writing an abstract interpreter we first design its parameters. The interpreter will be designed such that variations in these paramaters recover the concrete and a family of abstract interpretrs. To do this we extend the ideas developed in Van Horn and Might [2010] with a new parameter for path- and flow-sensitivity. When finished, we will be able to recover a concrete interpreter which respects the concrete semantics, and a family of abstract interpreters.

There will be three parameters to our abstract interpreter, one of which is novel in this work:

1. The monad, novel in this work. This is the execution engine of the interpreter and captures the path- and flow-sensitivity of the analysis.
2. The abstract domain. For our language this is merely the abstraction for integers.
3. Abstract Time. Abstract time captures the call-site sensitivity of the analysis.

For an object-oriented language, including a fourth parameter for object-sensitivity a la. Smaragdakis et al. [2011] is straightforward.

We place each of these parameters behind an abstract interface and leave their implementations opaque for the generic monadic interpreter. We will give each of these parameters reasoning principles as we introduce them. These principles allow us to reason about the correctness of the generic interpreter independent of a particular instantiation. The goal is to factor as much of the proof-effort into what we can say about the generic interpreter. An instantiation of the interpreter need only justify that each parameter meets their local interface.

4.1 The Analysis Monad

The monad for the interpreter captures the *effects* of interpretation. There are two effects we wish to model in the interpreter, state and nondeterminism. The state effect will mediate how the interpreter interacts with state cells in the state space, like Env and $Store$. The nondeterminism effect will mediate the branching of the execution from the interpreter. Our result is that path- and flow-sensitivities can be recovered by altering how these effects interact in the monad.

$$\begin{aligned} M &: \mathit{Type} \to \mathit{Type} \\ \mathit{bind} &: \forall \alpha\beta, M(\alpha) \to (\alpha \to M(\beta)) \to M(\beta) \\ \mathit{return} &: \forall \alpha, \alpha \to M(\alpha) \end{aligned}$$

Figure 2: Monad Interface

$$\begin{aligned} M &: \mathit{Type} \to \mathtt{type} \\ s &: \mathit{Type} \\ \mathit{get} &: M(s) \\ \mathit{put} &: s \to M(1) \end{aligned}$$

Figure 3: State Monad Interface

We briefly review monad, state and nondeterminism operators and their laws.

Base Monad Operations A type operator M is a monad if it support bind, a sequencing operator, and its unit return. The monad interface is summarized in Figure 2.

We use the monad laws to reason about our implementation in the absence of a particular implementation of bind and return:

$$\begin{aligned} \mathit{unit}_1 &: \mathit{bind}(\mathit{return}(a))(k) = k(a) \\ \mathit{unit}_2 &: \mathit{bind}(m)(\mathit{return}) = m \\ \mathit{assoc} &: \mathit{bind}(\mathit{bind}(m)(k_1))(k_2) \\ &\quad = \mathit{bind}(m)(\lambda(a).\, \mathit{bind}(k_1(a))(k_2)) \end{aligned}$$

bind and return mean something different for each monadic effect class. For state, bind is a sequencer of state and return is the "no change in state" effect. For nondeterminism, bind implements a merging of multiple branches and return is the singleton branch.

As is traditional with monadic programming, we use `do` and semicolon notation as syntactic sugar for bind. For example: $a \leftarrow m \ ; \ k(a)$ is just sugar for $\mathit{bind}(m)(k)$. We replace semicolons with line breaks headed by a `do` command for multiline monadic definitions.

Monadic State Operations A type operator M supports the monadic state effect for a type s if it supports get and put actions over s. The state monad interface is summarized in Figure 3.

We use the state monad laws to reason about state effects:

$$\begin{aligned} \mathit{put\text{-}put} &: \mathit{put}(s_1) \ ; \ \mathit{put}(s_2) = \mathit{put}(s_2) \\ \mathit{put\text{-}get} &: \mathit{put}(s) \ ; \ \mathit{get} = \mathit{return}(s) \\ \mathit{get\text{-}put} &: s \leftarrow \mathit{get} \ ; \ \mathit{put}(s) = \mathit{return}(1) \\ \mathit{get\text{-}get} &: s_1 \leftarrow \mathit{get} \ ; \ s_2 \leftarrow \mathit{get} \ ; \ k(s_1, s_2) = s \leftarrow \mathit{get} \ ; \ k(s, s) \end{aligned}$$

$$M : \; \mathit{Type} \rightarrow \mathit{Type}$$
$$\mathit{mzero} : \forall \alpha, M(\alpha)$$
$$\_ \langle + \rangle \_ : \forall \alpha, M(\alpha) \times M(\alpha) \rightarrow M(\alpha)$$

Figure 4: Nondeterminism Interface

$$\mathit{Val} : \; \mathit{Type}$$
$$\bot : \; \mathit{Val}$$
$$\_ \sqcup \_ : \; \mathit{Val} \times \mathit{Val} \rightarrow \mathit{Val}$$
$$\mathit{int\text{-}I} : \mathbb{Z} \rightarrow \mathit{Val}$$
$$\mathit{int\text{-}if0\text{-}E} : \; \mathit{Val} \rightarrow \mathcal{P}(\mathit{Bool})$$
$$\mathit{clo\text{-}I} : \; \mathit{Clo} \rightarrow \mathit{Val}$$
$$\mathit{clo\text{-}E} : \; \mathit{Val} \rightarrow \mathcal{P}(\mathit{Clo})$$
$$\delta[\![\_, \_, \_]\!] : \; \mathit{IOp} \times \mathit{Val} \times \mathit{Val} \rightarrow \mathit{Val}$$

Figure 5: Abstract Domain Interface

Nondeterminism Operations A type operator M support the nondeterminism effect if it supports an alternation operator $\langle + \rangle$ and its unit mzero. The nondeterminism interface is summarized in Figure **??**.

We use the nondeterminism laws to reason about nondeterminism effects:

$$\bot\text{-}\mathit{zero}_1 : \; \mathit{bind}(\mathit{mzero})(k) = \mathit{mzero}$$
$$\bot\text{-}\mathit{zero}_2 : \; \mathit{bind}(m)(\lambda(a).\, \mathit{mzero}) = \mathit{mzero}$$
$$\bot\text{-}\mathit{unit}_1 : \; \mathit{mzero} \langle + \rangle m = m$$
$$\bot\text{-}\mathit{unit}_2 : m \langle + \rangle \; \mathit{mzero} = m$$
$$+\text{-}\mathit{assoc} : m_1 \langle + \rangle (m_2 \langle + \rangle m_3) = (m_1 \langle + \rangle m_2) \langle + \rangle m_3$$
$$+\text{-}\mathit{comm} : m_1 \langle + \rangle m_2 = m_2 \langle + \rangle m_1$$
$$+\text{-}\mathit{dist} :$$
$$\mathit{bind}(m_1 \langle + \rangle m_2)(k) = \mathit{bind}(m_1)(k) \langle + \rangle \mathit{bind}(m_2)(k)$$

Together, all the monadic operators we have shown capture the essence of combining explicit state-passing and set comprehension. Our interpreter will use these operators and avoid referencing an explicit configuration ς or explicit collections of results.

4.2 The Abstract Domain

The abstract domain is encapsulated by the Val type in the semantics. To parameterize over it, we make Val opaque but require it support various operations. There is a constraint on Val its-self: it must be a join-semilattice with $\bot$ and $\sqcup$ respecting the usual laws. We require Val to be a join-semilattice so it can be merged in the Store. The interface for the abstract domain is shown in Figure 5.

$$\mathit{Time} : \; \mathit{Type}$$
$$\mathit{tick} : \; \mathit{Exp} \times \mathit{KAddr} \times \mathit{Time} \rightarrow \mathit{Time}$$

Figure 6: Abstract Time Interface

The laws for this interface are designed to induce a Galois connection between $\mathbb{Z}$ and Val:

$$\{\mathbf{true}\} \sqsubseteq \mathit{int\text{-}if0\text{-}E}(\mathit{int\text{-}I}(i)) \; \texttt{if} \; i = 0$$
$$\{\mathbf{false}\} \sqsubseteq \mathit{int\text{-}if0\text{-}E}(\mathit{int\text{-}I}(i)) \; \texttt{if} \; i \neq 0$$
$$v \sqsupseteq \bigsqcup_{b \in \mathit{int\text{-}if0\text{-}E}(v)} \theta(b)$$

where

$$\theta(\mathbf{true}) = \mathit{int\text{-}I}(0)$$
$$\theta(\mathbf{false}) = \bigsqcup_{i \in \mathbb{Z} \,|\, i \neq 0} \mathit{int\text{-}I}(i)$$

Closures must follow similar laws:

$$\{c\} \sqsubseteq \mathit{clo\text{-}E}(\mathit{cloI}(c))$$
$$v \sqsubseteq \bigsqcup_{c \in \mathit{clo\text{-}E}(v)} \mathit{clo\text{-}I}(c)$$

And δ must be sound w.r.t. the abstract semantics:

$$\mathit{int\text{-}I}(i_1 + i_2) \sqsubseteq \delta[\![+, \mathit{int\text{-}I}(i_1), \mathit{int\text{-}I}(i_2)]\!]$$
$$\mathit{int\text{-}I}(i_1 - i_2) \sqsubseteq \delta[\![-, \mathit{int\text{-}I}(i_1), \mathit{int\text{-}I}(i_2)]\!]$$

Supporting additional primitive types like booleans, lists, or arbitrary inductive datatypes is analogous. Introduction functions inject the type into Val. Elimination functions project a finite set of discrete observations. Introduction and elimination operators must follow a Galois connection discipline.

Of note is our restraint from allowing operations over Val to have monadic effects. We set things up specifically in this way so that Val and the monad M can be varied independent of each other.

4.3 Abstract Time

The interface for abstract time is familiar from Abstracting Abstract Machines [Van Horn and Might 2010](AAM)–which introduces abstract time as a single parameter from variations in call-site sensitivity–and is shown in Figure 6. In AAM, tick is defined to have access to all of Σ. This comes from the generality of the framework–to account for all possible tick functions. We only discuss instantiating Addr to support k-CFA, so we specialize the Σ parameter to $\mathit{Exp} \times \mathit{KAddr}$. Also in AAM is the opaque function $\mathit{alloc} : \; \mathit{Var} \times \mathit{Time} \; \rightarrow \; \mathit{Addr}$. Because we will only ever use the identity function for alloc, we omit its abstraction and instantiation in our development.

Remarkably, we need not state laws for tick. Our interpreter will always merge values which reside at the same

$$
\begin{aligned}
&A[\![\_]\!] \in \mathit{Atom} \to M(\mathit{Val})\\
&A[\![i]\!] \coloneqq \mathit{return}(\mathit{int\text{-}I}(i))\\
&A[\![x]\!] \coloneqq \texttt{do}\\
&\quad \rho \leftarrow \mathit{get\text{-}Env}\\
&\quad \sigma \leftarrow \mathit{get\text{-}Store}\\
&\quad l \leftarrow \uparrow_p(\rho(x))\\
&\quad \mathit{return}(\sigma(x))\\
&A[\![\underline{\boldsymbol{\lambda}}(x).e]\!] \coloneqq \texttt{do}\\
&\quad \rho \leftarrow \mathit{get\text{-}Env}\\
&\quad \mathit{return}(\mathit{clo\text{-}I}(\langle \underline{\boldsymbol{\lambda}}(x).e, \rho\rangle))
\end{aligned}
$$

Figure 7: Monadic denotation for atoms

address to achieve soundness. Therefore, any supplied implementations of tick is valid.

5. The Interpreter

We now present a generic monadic interpreter for $\boldsymbol{\lambda}$**IF** parameterized over M, Val and Time.

First we implement $A[\![\_]\!]$, a *monadic* denotation for atomic expressions, shown in Figure 7.

$\mathit{get\text{-}Env}$ and $\mathit{get\text{-}Store}$ are primitive operations for monadic state. $\mathit{clo\text{-}I}$ comes from the abstract domain interface. $\uparrow_p$ is the lifting of values from powerset into the monad:

$$
\begin{aligned}
&\uparrow_p : \forall \alpha, \mathcal{P}(\alpha) \to M(\alpha)\\
&\uparrow_p(\{a_1..a_n\}) \coloneqq \mathit{return}(a_1) \langle + \rangle .. \langle + \rangle \mathit{return}(a_n)
\end{aligned}
$$

Next we implement step, a *monadic* small-step function for compound expressions, shown in Figure 8. step uses helper functions push and pop for manipulating stack frames:

$$
\begin{aligned}
&\mathit{push} : \ \mathit{Frame} \to M(1)\\
&\mathit{push}(\mathit{fr}) \coloneqq \texttt{do}\\
&\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\\
&\quad \kappa\sigma \leftarrow \mathit{get\text{-}KStore}\\
&\quad \kappa l' \leftarrow \mathit{get\text{-}Time}\\
&\quad \mathit{put\text{-}KStore}(\kappa\sigma \sqcup [\kappa l' \mapsto \{\mathit{fr} :: \kappa l\}])\\
&\quad \mathit{put\text{-}KAddr}(\kappa l')\\
&\mathit{pop} : \ M(\mathit{Frame})\\
&\mathit{pop} \coloneqq \texttt{do}\\
&\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\\
&\quad \kappa\sigma \leftarrow \mathit{get\text{-}KStore}\\
&\quad \mathit{fr} :: \kappa l' \leftarrow \uparrow_p(\kappa\sigma(\kappa l))\\
&\quad \mathit{put\text{-}KAddr}(\kappa l')\\
&\quad \mathit{return}(\mathit{fr})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{step} : \ \mathit{Exp} \to M(\mathit{Exp})\\
&\mathit{step}(e_1 \odot e_2) \coloneqq \texttt{do}\\
&\quad \mathit{tickM}(e_1 \odot e_2)\\
&\quad \mathit{push}(\langle \square \odot e_2\rangle)\\
&\quad \mathit{return}(e_1)\\
&\mathit{step}(a) \coloneqq \texttt{do}\\
&\quad \mathit{tickM}(a)\\
&\quad \mathit{fr} \leftarrow \mathit{pop}\\
&\quad v \leftarrow A[\![a]\!]\\
&\quad \texttt{case}\ \mathit{fr}\ \texttt{of}\\
&\qquad \langle \square \odot e\rangle \to \texttt{do}\\
&\qquad\quad \mathit{push}(\langle v \odot \square\rangle)\\
&\qquad\quad \mathit{return}(e)\\
&\qquad \langle v' \ \texttt{@}\ \square\rangle \to \texttt{do}\\
&\qquad\quad \langle \underline{\boldsymbol{\lambda}}(x).e, \rho'\rangle \leftarrow \uparrow_p(\mathit{clo\text{-}E}(v'))\\
&\qquad\quad \tau \leftarrow \mathit{get\text{-}Time}\\
&\qquad\quad \sigma \leftarrow \mathit{get\text{-}Store}\\
&\qquad\quad \mathit{put\text{-}Env}(\rho'[x \mapsto (x,\tau)])\\
&\qquad\quad \mathit{put\text{-}Store}(\sigma \sqcup [(x,\tau) \mapsto \{v\}])\\
&\qquad\quad \mathit{return}(e)\\
&\qquad \langle v' \oplus \square\rangle \to \texttt{do}\\
&\qquad\quad \mathit{return}(\delta(\oplus, v', v))\\
&\qquad \langle \underline{\mathbf{if0}}(\square)\{e_1\}\{e_2\}\rangle \to \texttt{do}\\
&\qquad\quad b \leftarrow \uparrow_p(\mathit{int\text{-}if0\text{-}E}(v))\\
&\qquad\quad \texttt{if}(b)\ \ \texttt{then}\ \ \mathit{return}(e_1)\ \ \texttt{else}\ \ \mathit{return}(e_2)
\end{aligned}
$$

Figure 8: Monadic step function

and a monadic version of tick called tickM:

$$
\begin{aligned}
&\mathit{tickM} : \ \mathit{Exp} \to M(1)\\
&\mathit{tickM}(e) = \texttt{do}\\
&\quad \tau \leftarrow \mathit{get\text{-}Time}\\
&\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\\
&\quad \mathit{put\text{-}Time}(\mathit{tick}(e, \kappa l, \tau))
\end{aligned}
$$

We can also implement abstract garbage collection in a fully general away against the monadic effect interface:

$$
\begin{aligned}
&\mathit{gc} : \ \mathit{Exp} \to M(1)\\
&\mathit{gc}(e) \coloneqq \texttt{do}\\
&\quad \rho \leftarrow \mathit{get\text{-}Env}\\
&\quad \sigma \leftarrow \mathit{get\text{-}Store}\\
&\quad \kappa\sigma \leftarrow \mathit{get\text{-}KStore}\\
&\quad \mathit{put\text{-}Store}(\{l \mapsto \sigma(l) \mid l \in R[\sigma](\rho, e))\\
&\quad \mathit{put\text{-}KStore}(\{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in \mathit{KR}[\kappa\sigma](\kappa l)\})
\end{aligned}
$$

where R and KR are as defined in Section 2. The interpreter looks deterministic, however the nondeterminism is abstracted away behind $\uparrow_p$ and monadic bind.

In generalizing the semantics to account for nondeterminism, updates to both the value and continuation store must merge rather than strong update. This is because we place no restriction on the semantics for $Time$, and we must preserve soundness in the presence of reused addresses. Our interpreter is therefore operating over a modified state space:

$$\sigma \in \mathit{Store} : \mathit{Addr} \rightarrow \mathit{Val}$$
$$\kappa\sigma \in \mathit{KStore} : \mathit{KAddr} \rightarrow \mathcal{P}(\mathit{Frame} \times \mathit{KAddr})$$

We have already established a join-semilattice structure in the interface for Val in the abstract domain interface. Developing a custom join-semilattice for continuations is possible, and is the key component of recent developments in pushdown abstraction. For this presentation we use $\mathcal{P}(\mathit{Frame} \times \mathit{KAddr})$ as an abstraction for continuations for simplicity.

To execute the interpreter we must introduce one more parameter. In the concrete semantics, execution takes the form of a least-fixed-point computation over the collecting semantics This in general requires a join-semilattice structure for some Σ and a transition function $\Sigma \rightarrow \Sigma$. We bridge this gap between monadic interpreters and transition functions with an extra constraint on the monad M. We require that monadic actions $\mathit{Exp} \rightarrow M(\mathit{Exp})$ form a Galois connection with a transition system $\Sigma \rightarrow \Sigma$. This Galois connection serves two purposes. First, it allows us to implement the analysis by converting our interpreter to the transition system $\Sigma \rightarrow \Sigma$ through γ. Second, this Galois connection serves to *transport other Galois connections* as part of our correctness framework. For example, given concrete and abstract versions of Val, we carry $\mathbf{Val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Val}}$ through the Galois connection to establish $\mathbf{\Sigma} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{\Sigma}}$.

A collecting-semantics execution of our interpreter is defined as the least-fixed-point of $step$ transported through the Galois connection.

$$\mu(X).\varsigma_0 \sqcup X \sqcup \gamma(\mathit{step})(X)$$

where ς_0 is the injection of the initial program e_0 into Σ.

6. Recovering Analyses

To recover concrete and abstract interpreters we need only instantiate our generic monadic interpreter with concrete and abstract components.

6.1 Recovering a Concrete Interpreter

For the concrete value space we instantiate Val to $\mathbf{Val}$, a powerset of values:

$$v \in \mathbf{Val} := \mathcal{P}(\mathbf{Clo} + \mathbb{Z})$$

The concrete value space $\mathbf{Val}$ has straightforward introduction and elimination rules:

$$\mathit{int\text{-}I} : \mathbb{Z} \rightarrow \mathbf{Val}$$
$$\mathit{int\text{-}I}(i) := \{i\}$$
$$\mathit{int\text{-}if0\text{-}E} : \mathbf{Val} \rightarrow \mathcal{P}(\mathit{Bool})$$
$$\mathit{int\text{-}if0\text{-}E}(v) := \{\mathbf{true} \mid 0 \in v\} \cup \{\mathbf{false} \mid i \in v \wedge i \neq 0\}$$

and the concrete δ you would expect:

$$\delta[\![\_,\_,\_]\!] : \mathit{IOp} \times \mathbf{Val} \times \mathbf{Val} \rightarrow \mathbf{Val}$$
$$\delta[\![+, v_1, v_2]\!] := \{i_1 + i_2 \mid i_1 \in v_1 \; ; \; i_2 \in v_2\}$$
$$\delta[\![-, v_1, v_2]\!] := \{i_1 - i_2 \mid i_1 \in v_1 \; ; \; i_2 \in v_2\}$$

Proposition 1. $\mathbf{Val}$ *satisfies the abstract domain laws shown in Section 4.2 Figure 5.*

Concrete time $\mathbf{Time}$ captures program contours as a product of Exp and $\mathbf{KAddr}$:

$$\tau \in \mathbf{Time} := (\mathit{Exp} \times \mathit{KAddr})^*$$

and $tick$ is just a cons operator:

$$\mathit{tick} : \mathit{Exp} \times \mathbf{KAddr} \times \mathbf{Time} \rightarrow \mathbf{Time}$$
$$\mathit{tick}(e, \kappa l, \tau) := (e, \kappa l) :: \tau$$

For the concrete monad we instantiate M to a path-sensitive $\mathbf{M}$ which contains a powerset of concrete state space components.

$$\psi \in \Psi := \mathbf{Env} \times \mathbf{Store} \times \mathbf{KAddr} \times \mathbf{KStore} \times \mathbf{Time}$$
$$m \in \mathbf{M}(\alpha) := \Psi \rightarrow \mathcal{P}(\alpha \times \Psi)$$

Monadic operators $bind$ and $return$ encapsulate both state-passing and set-flattening:

$$\mathit{bind} : \forall \alpha, \mathbf{M}(\alpha) \rightarrow (\alpha \rightarrow \mathbf{M}(\beta)) \rightarrow \mathbf{M}(\beta)$$
$$\mathit{bind}(m)(f)(\psi) :=$$
$$\{(y, \psi'') \mid (y, \psi'') \in f(a)(\psi') \; ; \; (a, \psi') \in m(\psi)\}$$
$$\mathit{return} : \forall \alpha, \alpha \rightarrow \mathbf{M}(\alpha)$$
$$\mathit{return}(a)(\psi) := \{(a, \psi)\}$$

State effects merely return singleton sets:

$$\mathit{get\text{-}Env} : \mathbf{M}(\mathbf{Env})$$
$$\mathit{get\text{-}Env}(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(\rho, \langle \rho, \sigma, \kappa, \tau \rangle)\}$$
$$\mathit{put\text{-}Env} : \mathbf{Env} \rightarrow \mathcal{P}(1)$$
$$\mathit{put\text{-}Env}(\rho')(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(1, \langle \rho', \sigma, \kappa, \tau \rangle)\}$$

Nondeterminism effects are implemented with set union:

$$\mathit{mzero} : \forall \alpha, \mathbf{M}(\alpha)$$
$$\mathit{mzero}(\psi) := \{\}$$
$$\_ \langle + \rangle \_ : \forall \alpha, \mathbf{M}(\alpha) \times \mathbf{M}(\alpha) \rightarrow \mathbf{M}(\alpha)$$
$$(m_1 \langle + \rangle m_2)(\psi) := m_1(\psi) \cup m_2(\psi)$$

Proposition 2. $\mathbf{M}$ *satisfies monad, state, and nondeterminism laws shown in Section 4.1 Figures 2, 3 and 4.*

Finally, we must establish a Galois connection between $Exp \rightarrow \mathbf{M}(Exp)$ and $\mathbf{\Sigma} \rightarrow \mathbf{\Sigma}$ for some choice of $\mathbf{\Sigma}$. For the path-sensitive monad $\mathbf{M}$ instantiate with $\mathbf{Val}$ and $\mathbf{Time}$, , $\mathbf{\Sigma}$ is defined:

$$\mathbf{\Sigma} \coloneqq \mathcal{P}(Exp \times \Psi)$$

The Galois connection between $\mathbf{M}$ and $\mathbf{\Sigma}$ is straightforward:

$$\begin{aligned}
&\gamma\colon (Exp \rightarrow \mathbf{M}(Exp)) \rightarrow (\mathbf{\Sigma} \rightarrow \mathbf{\Sigma})\\
&\gamma(f)(e\psi*) \coloneqq \{(e, \psi') \mid (e, \psi') \in f(e)(\psi) \; ; \; (e, \psi) \in e\psi*\}\\
&\alpha\colon (\mathbf{\Sigma} \rightarrow \mathbf{\Sigma}) \rightarrow (Exp \rightarrow \mathbf{M}(Exp))\\
&\alpha(f)(e)(\psi) \coloneqq f(\{(e, \psi)\})
\end{aligned}$$

The injection ς_0 for a program e_0 is:

$$\varsigma_0 \coloneqq \{\langle e, \bot, \bot, , \bot, \rangle\}$$

Proposition 3. *γ and α form an isomorphism.*

Corollary 1. *γ and α form a Galois connection.*

6.2 Recovering an Abstract Interpreter

We pick a simple abstraction for integers, $\{-, 0, +\}$, although our technique scales seamlessly to other domains.

$$\widehat{\mathbf{Val}} \coloneqq \mathcal{P}(\widehat{\mathbf{Clo}} + \{-, 0, +\})$$

Introduction and elimination functions for $\widehat{\mathbf{Val}}$ are defined:

$$\begin{aligned}
&\mathit{int\text{-}I}\colon \mathbb{Z} \rightarrow \widehat{\mathbf{Val}}\\
&\mathit{int\text{-}I}(i) \coloneqq - \texttt{ if } i < 0\\
&\mathit{int\text{-}I}(i) \coloneqq 0 \texttt{ if } i = 0\\
&\mathit{int\text{-}I}(i) \coloneqq + \texttt{ if } i > 0\\
&\mathit{int\text{-}if0\text{-}E}\colon \widehat{\mathbf{Val}} \rightarrow \mathcal{P}(Bool)\\
&\mathit{int\text{-}if0\text{-}E}(v) \coloneqq \{\mathbf{true} \mid 0 \in v\} \cup \{\mathbf{false} \mid - \in v \vee + \in v\}
\end{aligned}$$

Introduction and elimination for $\widehat{\mathbf{Clo}}$ is identical to the concrete domain.

The abstract δ operator is defined:

$$\begin{aligned}
&\delta\colon \; IOp \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Val}} \rightarrow \widehat{\mathbf{Val}}\\
&\delta(+, v_1, v_2) \coloneqq\\
&\qquad \{i \mid 0 \in v_1 \wedge i \in v_2\}\\
&\quad \cup \{i \mid i \in v_1 \wedge 0 \in v_2\}\\
&\quad \cup \{+ \mid + \in v_1 \wedge + \in v_2\}\\
&\quad \cup \{- \mid - \in v_1 \wedge - \in v_2\}\\
&\quad \cup \{-, 0, + \mid + \in v_1 \wedge - \in v_2\}\\
&\quad \cup \{-, 0, + \mid - \in v_1 \wedge + \in v_2\}
\end{aligned}$$

The definition for $\delta(-, v_1, v_2)$ is analogous.

Proposition 4. *$\widehat{\mathbf{Val}}$ satisfies the abstract domain laws shown in Section 4.2 Figure 5.*

Proposition 5. $\mathbf{Val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Val}}$ *and their operations $\mathit{int\text{-}I}$, $\mathit{int\text{-}if0\text{-}E}$ and δ are ordered $\sqsubseteq$ respectively through the Galois connection.*

Next we abstract $Time$ to $\widehat{\mathbf{Time}}$ as the finite domain of k-truncated lists of execution contexts:

$$\widehat{\mathbf{Time}} \coloneqq (Exp \times \widehat{\mathbf{KAddr}})^*_k$$

The *tick* operator becomes cons followed by k-truncation:

$$\begin{aligned}
&\mathit{tick}\colon \; Exp \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{Time}} \rightarrow \widehat{\mathbf{Time}}\\
&\mathit{tick}(e, \kappa l, \tau) = \lfloor (e, \kappa l) :: \tau \rfloor_k
\end{aligned}$$

Proposition 6. $\mathbf{Time} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Time}}$ *and tick is ordered $\sqsubseteq$ through the Galois connection.*

The monad $\widehat{\mathbf{M}}$ need not change in implementation from $\mathbf{M}$; they are identical up the choice of Ψ.

$$\psi \in \Psi \coloneqq \widehat{\mathbf{Env}} \times \widehat{\mathbf{Store}} \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{KStore}} \times \widehat{\mathbf{Time}}$$

The resulting state space $\widehat{\mathbf{\Sigma}}$ is finite, and its least-fixed-point iteration will give a sound and computable analysis.

7. Varying Path- and Flow-Sensitivity

We are able to recover a flow-insensitivity in the analysis through a new definition for M: $\widehat{\mathbf{M}}^{fi}$. To do this we pull $\widehat{\mathbf{Store}}$ out of the powerset, exploiting its join-semilattice structure:

$$\begin{aligned}
&\Psi \coloneqq \widehat{\mathbf{Env}} \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{KStore}} \times \widehat{\mathbf{Time}}\\
&\widehat{\mathbf{M}}^{fi}(\alpha) \coloneqq \Psi \times \widehat{\mathbf{Store}} \rightarrow \mathcal{P}(\alpha \times \Psi) \times \widehat{\mathbf{Store}}
\end{aligned}$$

The monad operator *bind* performs the store merging needed to capture a flow-insensitive analysis.

$$\begin{aligned}
&\mathit{bind}\colon \forall \alpha\beta, \widehat{\mathbf{M}}^{fi}(\alpha) \rightarrow (\alpha \rightarrow \widehat{\mathbf{M}}^{fi}(\beta)) \rightarrow \widehat{\mathbf{M}}^{fi}(\beta)\\
&\mathit{bind}(m)(f)(\psi, \sigma) \coloneqq (\{bs_{11}..bs_{n1}..bs_{nm}\}, \sigma_1 \sqcup .. \sqcup \sigma_n)\\
&\quad \text{where}\\
&\qquad (\{(a_1, \psi_1)..(a_n, \psi_n)\}, \sigma') \coloneqq m(\psi, \sigma)\\
&\qquad (\{b\psi_{i1}..b\psi_{im}\}, \sigma_i) \coloneqq f(a_i)(\psi_i, \sigma')
\end{aligned}$$

The unit for *bind* returns one nondeterminism branch and a single store:

$$\begin{aligned}
&\mathit{return}\colon \forall \alpha, \alpha \rightarrow \widehat{\mathbf{M}}^{fi}(\alpha)\\
&\mathit{return}(a)(\psi, \sigma) \coloneqq (\{a, \psi\}, \sigma)
\end{aligned}$$

State effects *get-Env* and *put-Env* are also straightforward, returning one branch of nondeterminism:

$$\begin{aligned}
&\mathit{get\text{-}Env}\colon \widehat{\mathbf{M}}^{fi}(\widehat{\mathbf{Env}})\\
&\mathit{get\text{-}Env}(\langle \rho, \kappa, \tau \rangle, \sigma) \coloneqq (\{(\rho, \langle \rho, \kappa, \tau \rangle)\}, \sigma)\\
&\mathit{put\text{-}Env}\colon \widehat{\mathbf{Env}} \rightarrow \widehat{\mathbf{M}}^{fi}(1)\\
&\mathit{put\text{-}Env}(\rho')(\langle \rho, \kappa, \tau \rangle, \sigma) \coloneqq (\{(1, \langle \rho', \kappa, \tau \rangle)\}, \sigma)
\end{aligned}$$

State effects $get\text{-}Store$ and $put\text{-}Store$ are analogous to $get\text{-}Env$ and $put\text{-}Env$:

$$get\text{-}Store : \widehat{\mathbf{M}}^{fi}(\widehat{\mathbf{Env}})$$
$$get\text{-}Store(\langle \rho, \kappa, \tau \rangle, \sigma) := (\{(\sigma, \langle \rho, \kappa, \tau \rangle\}, \sigma)$$
$$put\text{-}Store : \widehat{\mathbf{Store}} \to \widehat{\mathbf{M}}^{fi}(1)$$
$$put\text{-}Store(\sigma')(\langle \rho, \kappa, \tau \rangle, \sigma) := (\{(1, \langle \rho, \kappa, \tau \rangle)\}, \sigma')$$

Nondeterminism operations will union the powerset and join the store pairwise:

$$mzero : \forall \alpha, M(\alpha)$$
$$mzero(\psi, \sigma) := (\{\}, \bot)$$
$$\_ \langle + \rangle \_ : \forall \alpha, M(\alpha) \times M(\alpha) \to M\ \alpha$$
$$(m_1 \langle + \rangle m_2)(\psi, \sigma) := (a\psi *_1 \cup a\psi *_2, \sigma_1 \sqcup \sigma_2)$$
$$\text{where } \ (a\psi *_i, \sigma_i) := m_i(\psi, \sigma)$$

Finally, the Galois connection relating $\widehat{\mathbf{M}}^{fi}$ to a state space transition over $\widehat{\mathbf{\Sigma}}^{fi}$ must also compute set unions and store joins pairwise:

$$\widehat{\mathbf{\Sigma}}^{fi} := \mathcal{P}(Exp \times \Psi) \times \widehat{\mathbf{Store}}$$
$$\gamma : (Exp \to \widehat{\mathbf{M}}^{fi}(Exp)) \to (\widehat{\mathbf{\Sigma}}^{fi} \to \widehat{\mathbf{\Sigma}}^{fi})$$
$$\gamma(f)(e\psi *, \sigma) := (\{e\psi_{11}..e\psi_{n1}..e\psi_{nm}\}, \sigma_1 \sqcup .. \sqcup \sigma_n)$$
$$\text{where}$$
$$\{(e_1, \psi_1)..(e_n, \psi_n)\} := e\psi *$$
$$(\{e\psi_{i1}..e\psi_{im}\}, \sigma_i) := f(e_i)(\psi_i, \sigma)$$
$$\alpha : (\widehat{\mathbf{\Sigma}}^{fi} \to \widehat{\mathbf{\Sigma}}^{fi}) \to (Exp \to \widehat{\mathbf{M}}^{fi}(Exp))$$
$$\alpha(f)(e)(\psi, \sigma) := f(\{(e, \psi)\}, \sigma)$$

Proposition 7. *γ and α form an isomorphism.*

Corollary 2. *γ and α form a Galois connection.*

Proposition 8. *There exists Galois connections:*

$$\mathbf{M} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \widehat{\mathbf{M}} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \widehat{\mathbf{M}}^{fi}$$

The first Galois connection $\mathbf{M} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \widehat{\mathbf{M}}$ is justified by the Galois connections between $\mathbf{Val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Val}}$ and $\mathbf{Time} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Time}}$. The second Galois connection $\widehat{\mathbf{M}} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \widehat{\mathbf{M}}^{fi}$ is justified by calculation over their definitions. We aim to recover this proof more easily through compositional components in Section 8.

Corollary 3.

$$\mathbf{\Sigma} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \widehat{\mathbf{\Sigma}} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \widehat{\mathbf{\Sigma}}^{fi}$$

This property is derived by transporting each Galois connection between monads through their respective Galois connections to Σ.

Proposition 9. *The following orderings hold between the three induced transition relations:*

$$\alpha_1 \circ \gamma(step) \circ \gamma_1 \sqsubseteq \widehat{\gamma}(step) \sqsubseteq \gamma_2 \circ \widehat{\gamma}^{fi}(step) \circ \alpha_2$$

This is a direct consequence of the monotonicity of step and the Galois connections between monads.

We note that the implementation for our interpreter and abstract garbage collector remain the same for each interpreter. They scale seamlessly to flow-sensitive and flow-insensitive variants when instantiated with the appropriate monad.

8. A Compositional Monadic Framework

In our development thus far, any modification to the interpreter requires redesigning the monad $\widehat{\mathbf{M}}$ and constructing new proofs. We want to avoid reconstructing complicated monads for our interpreters, especially as languages and analyses grow and change. Even more, we want to avoid reconstructing complicated *proofs* that such changes will necessarily alter. Toward this goal we introduce a compositional framework for constructing monads which are correct-by-construction. To do this we extend the well-known structure of monad transformer that that of *Galois transformer*.

There are two types of monadic effects used in our monadic interpreter: state and nondeterminism. Each of these effects have corresponding monad transformers. Our definition of a monad transformer for nondeterminism is novel in this work.

In the proceeding definitions, we must necessarily use $bind$, $return$, and other operations from the underlying monad. We notate these $bind_m$, $return_m$, $\texttt{do}_m$, $\leftarrow_m$, etc. for clarity.

8.1 State Monad Transformer

Briefly we review the state monad transformer, $S_t[s]$:

$$S_t[\_] : (Type \to Type) \to (Type \to Type)$$
$$S_t[s](m)(\alpha) := s \to m(\alpha \times s)$$

The state monad transformer can transport monadic operations from m to $S_t[s](m)$:

$$bind : \forall \alpha\beta, S_t[s](m)(\alpha) \to (\alpha \to S_t[s](m)(\beta)) \to S_t[s](m)(\beta)$$
$$bind(m)(f)(s) := \texttt{do}_m$$
$$(x, s') \leftarrow_m m(s)$$
$$f(x)(s')$$
$$return : \forall \alpha m, \alpha \to S_t[s](m)(\alpha)$$
$$return(x)(s) := return_m(x, s)$$

The state monad transformer can also transport nondeterminism effects from m to $S_t[s](m)$:

$$mzero : \forall \alpha, S_t[s](m)(\alpha)$$
$$mzero(s) := mzero_m$$
$$\_ \langle + \rangle \_ : \forall \alpha, S_t[s](m)(\alpha) \times S_t[s](m)(\alpha) \to S_t[s](m)(\alpha)$$
$$(m_1 \langle + \rangle m_2)(s) := m_1(s) \langle + \rangle_m m_2(s)$$

Finally, the state monad transformer exposes get and put operations given that m is a monad:

$$\begin{aligned}&get\colon S_t[s](m)(s)\\&get(s) \coloneqq return_m(s,s)\\&put\colon s \to S_t[s](m)(1)\\&put(s')(s) \coloneqq return_m(1,s')\end{aligned}$$

8.2 Nondeterminism Monad Transformer

We have developed a new monad transformer for nondeterminism which composes with state in both directions. Previous attempts to define a monad transformer for nondeterminism have resulted in monad operations which do not respect monad laws.

Our nondeterminism monad transformer shares the "expected" type, embedding $\mathcal{P}$ inside m:

$$\begin{aligned}&\mathcal{P}_t\colon (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})\\&\mathcal{P}_t(m)(\alpha) \coloneqq m(\mathcal{P}(\alpha))\end{aligned}$$

The nondeterminism monad transformer can transport monadic operations from m to $\mathcal{P}_t$ *provided that* m *is also a join-semilattice functor*:

$$\begin{aligned}&bind\colon \forall\alpha\beta, \mathcal{P}_t(m)(\alpha) \to (\alpha \to \mathcal{P}_t(m)(\beta)) \to \mathcal{P}_t(m)(\beta)\\&bind(m)(f) \coloneqq \mathtt{do}_m\\&\quad \{x_1..x_n\} \leftarrow_m m\\&\quad f(x_1) \sqcup_m .. \sqcup_m f(x_n)\\&return\colon \forall\alpha, \alpha \to \mathcal{P}_t(m)(\alpha)\\&return(x) \coloneqq return_m(\{x\})\end{aligned}$$

Proposition 10. *bind and return satisfy the monad laws.*

The key lemma in this proof is the functorality of m, namely that:

$$return_m(x \sqcup y) = return_m(x) \sqcup return_m(y)$$

The nondeterminism monad transformer can transport state effects from m to $\mathcal{P}_t$:

$$\begin{aligned}&get\colon \mathcal{P}_t(m)(s)\\&get = map_m(\lambda(s).\{s\})(get_m)\\&put\colon s \to \mathcal{P}_t(m)(s)\\&put(s) = map_m(\lambda(1).\{1\})(put_m(s))\end{aligned}$$

Proposition 11. *get and put satisfy the state monad laws.*

The proof is by simple calculation.

Finally, our nondeterminism monad transformer exposes nondeterminism effects as a straightforward application of the underlying monad's join-semilattice functorality:

$$\begin{aligned}&mzero\colon \forall\alpha, \mathcal{P}_t(m)(\alpha)\\&mzero \coloneqq \bot_m\\&\_\langle+\rangle\_\colon \forall\alpha, \mathcal{P}_t(m)(\alpha) x \mathcal{P}_t(m)(\alpha) \to \mathcal{P}_t(m)(\alpha)\\&m_1 \langle+\rangle m_2 \coloneqq m_1 \sqcup_m m_2\end{aligned}$$

Proposition 12. *mzero and* $\langle+\rangle$ *satisfy the nondeterminism monad laws.*

The proof is trivial as a consequence of the underlying monad being a join-semilattice functor.

8.3 Mapping to State Spaces

Both our execution and correctness frameworks requires that monadic actions in M map to some state space transitions Σ. We extend the earlier statement of Galois connection to the transformer setting:

$$mstep\colon \forall\alpha\beta, (\alpha \to M(\beta)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\Sigma(\alpha) \to \Sigma(\beta))$$

Here M must map *arbitrary* monadic actions $\alpha \to M(\beta)$ to state space transitions for a state space *functor* $\Sigma(\_)$ We only show the γ sides of the mappings in this section, which allow one to execute the analyses.

For the state monad transformer $S_t[s]$ mstep is defined:

$$\begin{aligned}&mstep\text{-}\gamma\colon \forall\alpha\beta m,\\&\quad (\alpha \to S_t[s](m)(\beta)) \to (\Sigma_m(\alpha \times s) \to \Sigma_m(\beta \times s))\\&mstep\text{-}\gamma(f) \coloneqq mstep_m\gamma(\lambda(a,s).f(a)(s))\end{aligned}$$

For the nondeterminism transformer $\mathcal{P}_t$, mstep has two possible definitions. One where Σ is $\Sigma_m \circ \mathcal{P}$:

$$\begin{aligned}&mstep_1\gamma\colon \forall\alpha\beta m,\\&\quad (\alpha \to \mathcal{P}_t(m)(\beta)) \to (\Sigma_m(\mathcal{P}(\alpha)) \to \Sigma_m(\mathcal{P}(\beta)))\\&mstep_1\gamma(f) \coloneqq mstep_m\gamma(F)\\&\quad \text{where } F(\{x_1..x_n\}) = f(x_1) \langle+\rangle .. \langle+\rangle f(x_n))\end{aligned}$$

and one where Σ is $\mathcal{P} \circ \Sigma_m$:

$$\begin{aligned}&mstep_2\gamma\colon \forall\alpha\beta m,\\&\quad (\alpha \to \mathcal{P}_t(m)(\beta)) \to (\mathcal{P}(\Sigma_m(\alpha)) \to \mathcal{P}(\Sigma_m(\beta)))\\&mstep_2\gamma(f)(\{\varsigma_1..\varsigma_n\}) \coloneqq a\Sigma P_1 \cup .. \cup a\Sigma P_n\\&\quad \text{where}\\&\qquad commuteP\text{-}\gamma\colon \forall\alpha, \Sigma_m(\mathcal{P}(\alpha)) \to \mathcal{P}(\Sigma_m(\alpha))\\&\qquad a\Sigma P_i \coloneqq commuteP\text{-}\gamma(mstep_m\gamma(f)(\varsigma_i))\end{aligned}$$

The operation $commuteP$-γ must be defined for the underlying Σ_m. In general, $commuteP$ must form a Galois connection. However, this property exists for the identity monad, and is preserved by $S_t[s]$, the only monad we will compose $\mathcal{P}_t$ with in this work.

$$\begin{aligned}&commuteP\text{-}\gamma\colon \forall\alpha, \Sigma_m(\mathcal{P}(\alpha) \times s) \to \mathcal{P}(\Sigma_m(\alpha \times s))\\&commuteP\text{-}\gamma \coloneqq commuteP_m \circ map(F)\\&\quad \text{where}\\&\qquad F(\{\alpha_1..\alpha_n\}) = \{(\alpha_1,s)..(\alpha_n,s)\})\end{aligned}$$

Of all the γ mappings defined, the γ side of $commuteP$ is the only mapping that loses information in the α direction. Therefore, $mstep_{S_t[s]}$ and $mstep_{\mathcal{P}_t 1}$ are really isomorphism transformers, and $mstep_{\mathcal{P}_t 2}$ is the only Galois connection transformer. The Galois connections for $mstep$ for both $S_t[s]$ or P_t rely crucially on $mstep_m \gamma$ and $mstep_m \alpha$ be homomorphic, i.e. that:

$$\alpha(id) \sqsubseteq return$$
$$\alpha(f \circ g) \sqsubseteq \alpha(f) \langle \circ \rangle \alpha(g)$$

and likewise for γ, where $\langle \circ \rangle$ is composition in the Kleisli category for the monad M.

For convenience, we name the pairing of $\mathcal{P}_t$ with $mstep_1$ FI_t, and with $mstep_2$ FS_t for flow-insensitive and flow-sensitive respectively.

Proposition 13. $\Sigma_{FS_t} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Sigma_{FI_t}$.

The proof is by consequence of $commuteP$.

Proposition 14. $S_t[s] \circ \mathcal{P}_t \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}_t \circ S_t[s]$.

The proof is by calculation after unfolding the definitions.

8.4 Galois Transformers

The capstone of our compositional framework is the fact that monad transformers $S_t[s]$ and $\mathcal{P}_t$ are also *Galois transformers*. Whereas a monad transformer is a functor between functors, a Galois transformer is a functor between Galois functors.

Definition 1. *A monad transformer T is a Galois transformer if for Galois functors m_1 and m_2, $m_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} m_2 \implies T(m_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} T(m_2)$.*

Proposition 15. *$S_t[s]$ and $\mathcal{P}_t$ are Galois transformers.*

The proofs are straightforward applications of the underlying $m_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} m_2$.

Furthermore, the state monad transformer $S_t[s]$ is Galois functorial in its state parameter s.

8.5 Building Transformer Stacks

We can now build monad transformer stacks from combinations of $S_t[s]$, FI_t and FS_t that have the following properties:

- The resulting monad has the combined effects of all pieces of the transformer stack.
- Actions in the resulting monad map to a state space transition system $\Sigma \to \Sigma$ for some Σ.
- Galois connections between $\mathbf{\Sigma}$ and $\widehat{\mathbf{\Sigma}}$ are established piecewise from monad transformer components.
- Monad transformer components are proven correct once and for all.

We instantiate our interpreter to the following monad stacks in decreasing order of precision:

$$\begin{array}{lll}
S_t[\widehat{\mathbf{Env}}] & S_t[\widehat{\mathbf{Env}}] & S_t[\widehat{\mathbf{Env}}] \\
S_t[\widehat{\mathbf{KAddr}}] & S_t[\widehat{\mathbf{KAddr}}] & S_t[\widehat{\mathbf{KAddr}}] \\
S_t[\widehat{\mathbf{KStore}}] & S_t[\widehat{\mathbf{KStore}}] & S_t[\widehat{\mathbf{KStore}}] \\
S_t[\widehat{\mathbf{Time}}] & S_t[\widehat{\mathbf{Time}}] & S_t[\widehat{\mathbf{Time}}] \\
S_t[\widehat{\mathbf{Store}}] & FS_t & FI_t \\
FS_t & S_t[\widehat{\mathbf{Store}}] & S_t[\widehat{\mathbf{Store}}]
\end{array}$$

From left to right, these give path-sensitive, flow-sensitive, and flow-insensitive analyses. Furthermore, each monad stack with abstract components is assigned a Galois connection by-construction with their concrete analogues:

$$\begin{array}{lll}
S_t[\mathbf{Env}] & S_t[\mathbf{Env}] & S_t[\mathbf{Env}] \\
S_t[\mathbf{KAddr}] & S_t[\mathbf{KAddr}] & S_t[\mathbf{KAddr}] \\
S_t[\mathbf{KStore}] & S_t[\mathbf{KStore}] & S_t[\mathbf{KStore}] \\
S_t[\mathbf{Time}] & S_t[\mathbf{Time}] & S_t[\mathbf{Time}] \\
S_t[\mathbf{Store}] & FS_t & FI_t \\
FS_t & S_t[\mathbf{Store}] & S_t[\mathbf{Store}]
\end{array}$$

Another benefit of our approach is that we can selectively widen the value store and the continuation store independent of each other. To do this we merely swap the order of transformers:

$$\begin{array}{lll}
S_t[\widehat{\mathbf{Env}}] & S_t[\widehat{\mathbf{Env}}] & S_t[\widehat{\mathbf{Env}}] \\
S_t[\widehat{\mathbf{KAddr}}] & S_t[\widehat{\mathbf{KAddr}}] & S_t[\widehat{\mathbf{KAddr}}] \\
S_t[\widehat{\mathbf{Time}}] & S_t[\widehat{\mathbf{Time}}] & S_t[\widehat{\mathbf{Time}}] \\
S_t[\widehat{\mathbf{KStore}}] & FS_t & FI_t \\
S_t[\widehat{\mathbf{Store}}] & S_t[\widehat{\mathbf{KStore}}] & S_t[\widehat{\mathbf{KStore}}] \\
FS_t & S_t[\widehat{\mathbf{Store}}] & S_t[\widehat{\mathbf{Store}}]
\end{array}$$

yielding analyses which are flow-sensitive and flow-insensitive for both the continuation and value stores.

9. Implementation

We have implemented our framework in Haskell and applied it to compute analyses for $\boldsymbol{\lambda}$**IF**. Our implementation provides path-sensitivity, flow-sensitivity, and flow-insensitivity as a semantics-independent monad library. The code shares a striking resemblance with the math.

Our interpreter for $\boldsymbol{\lambda}$**IF** is parameterized as discussed in Section 4. We express a valid analysis with the following Haskell constraint:

$$\begin{array}{l}
\texttt{type}\ Analysis(\delta, \mu, m) :: Constraint = \\
\quad (AAM(\mu), Delta(\delta), AnalysisMonad(\delta, \mu, m))
\end{array}$$

Constraints $AAM(\mu)$ and $Delta(\delta)$ are interfaces for abstract time and the abstract domain.

The constraint $AnalysisMonad(m)$ requires only that m has the required effects[1]:

$$
\begin{aligned}
&\texttt{type } AnalysisMonad(\delta, \mu, m) :: Constraint = (\\
&\quad Monad(m(\delta, \mu)), \\
&\quad MonadNondeterminism(m(\delta, \mu)), \\
&\quad MonadState(Env(\mu))(m(\delta, \mu)), \\
&\quad MonadState(Store(\delta, \mu))(m(\delta, \mu)), \\
&\quad MonadState(Time(\mu, Call))(m(\delta, \mu)))
\end{aligned}
$$

Our interpreter is implemented against this interface and concrete and abstract interpreters are recovered by instantiating δ, μ and m.

Our implementation is publicly available and can be installed as a cabal package by executing:

```
cabal install maam
```

10. Related Work

Program analysis comes in many forms such as points-to [**?**], flow [Jones 1981], or shape analysis [**?**], and the literature is vast. (See Hind [2001]; **?** for surveys.) Much of the research has focused on developing families or frameworks of analyses that endow the abstraction with a number of knobs, levers, and dials to tune precision and compute efficiently (some examples include Shivers [1991]; Nielson and Nielson [1997]; Milanova et al. [2005]; Van Horn and Might [2010]; there are many more). These parameters come in various forms with overloaded meanings such as object [Milanova et al. 2005; Smaragdakis et al. 2011], context [Sharir and Pnueli 1981; Shivers 1991], path [Das et al. 2002], and heap [Van Horn and Might 2010] sensitivities, or some combination thereof [**?**].

These various forms can all be cast in the theory of abstraction interpretation of Cousot and Cousot [1977, 1979] and understood as computable approximations of an underlying concrete interpreter. Our work demonstrates that if this underlying concrete interpreter is written in monadic style, monad transformers are a useful way to organize and compose these various kinds of program abstractions in a modular and language-independent way.
% This work inspired by the combination of Cousot and Cousot's theory of abstract interpretation based on Galois connections [1977; 1979; 1999], **?**'s monad transformers for modular interpreters [**?**], and **?**'s monadic abstract interpreters [**?**].

? first demonstrated how monad transformers could be used to define building blocks for constructing (concrete) interpreters. Their interpreter monad $InterpM$ bears a strong resemblance to ours. We show this "building blocks" approach to interpreter construction extends to *abstract* interpreter construction, too, by using Galois transfomers. Moreover, we show that these monad transformers can be proved sound via a Galois connection to their concrete counterparts, ensuring the soundness of any stack built from sound blocks of Galois transformers. Soundness proofs of various forms of analysis are notoriously brittle with respect to language and analysis features. A reusable framework of Galois transformers offers a potential way forward for a modular metatheory of program analysis.

Cousot [1999] develops a "calculational approach" to analysis design whereby analyses are not designed and then verified *post facto* but rather derived by positing an abstraction and calculating it through the concrete interpreter using Galois connections. These calculations are done by hand. Our approach offers a limited ability to automate the calculation process by relying on monad transformers to combine different abstractions.

? first introduced Monadic Abstract Interpreters (MAI), in which interpreters are also written in monadic style and variations in analysis are recovered through new monad implementations. However, each monad in MAI is designed from scratch for a specific language to have specific analysis properties. The MAI work is analogous to monadic interpreter of **?**, in which the monad structure is monolithic and must be reconstructed for each new language feature. Our work extends the ideas in MAI in a way that isolates each parameter to be independent of others, similar to the approach of **?**. We factor out the monad as a truly semantics independent feature. This factorization reveals an orthogonal tuning knob for path- and flow-sensitivity. Even more, we give the user building blocks for constructing monads that are correct and give the desired properties by construction. Our framework is also motivated by the needs of reasoning formally about abstract interpreters, no mention of which is made in MAI.

We build directly on the work of Abstracting Abstract Machines (AAM) by Van Horn and Might [2010] in our parameterization of abstract time and call-site sensitivity. More notably, we follow the AAM philosophy of instrumenting a concrete semantics *first* and performing a systematic abstraction *second*. This greatly simplifies the Galois connection arguments during systematic abstraction. However, this is at the cost of proving that the instrumented semantics simulate the original concrete semantics.

11. Conclusion

We have shown that *Galois transfomers*, monad transfomers that form Galois connections, are effective, language-independent building blocks for constructing program analyzers and form the basis of a modular, reusable, and composable metatheory for program analysis.

In the end, we hope language independent characterizations of analysis ingredients will both facilate the systematic construction of program analyses and bridge the gap between various communities which often work in isolation,

[1] We use a CPS representation and a single store in our implementation. This requires $Time$, which is generic to the language, to take $Call$ as a parameter rather than $Exp \times KAddr$.

despite the fruitful results of mapping between langauge paradigms such as the work of Might et al. [2010], showing that object-oriented k-CFA can be applied to functional languages to avoid the exponential time lower bound [Van Horn and Mairson 2008].

References

P. Cousot. The calculational design of a generic abstract interpreter. NATO ASI Series F. IOS Press, Amsterdam, 1999.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM, 1977.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. POPL '79. ACM, 1979.

M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. PLDI '02. ACM, 2002.

M. Hind. Pointer analysis: haven't we solved this problem yet? ACM, 2001.

N. D. Jones. Flow analysis of lambda expressions (preliminary version). Springer-Verlag, 1981.

M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP'06*, 2006.

M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. ÀCM, 2010.

A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 2005.

F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. ACM Press, 1997.

M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, chapter 7. Prentice-Hall, Inc., 1981.

O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. POPL '11. ACM, 2011.

D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. ACM, 2008.

D. Van Horn and M. Might. Abstracting abstract machines. ICFP '10. ACM, 2010.