

Galois Transformers and Modular Abstract Interpreters

Reusable Metatheory for Program Analysis

David Darais

University of Maryland, USA
darais@cs.umd.edu

Matthew Might

University of Utah, USA
might@cs.utah.edu

David Van Horn

University of Maryland, USA
dvanhorn@cs.umd.edu

Abstract

The design and implementation of static analyzers has become increasingly systematic. Yet for a given language or analysis feature, it often requires tedious and error prone work to implement an analyzer and prove it sound. In short, static analysis features and their proofs of soundness do not compose well, causing a dearth of reuse in both implementation and metatheory.

We solve the problem of systematically constructing static analyzers by introducing *Galois transformers*: monad transformers that transport Galois connection properties. In concert with a monadic interpreter, we define a library of monad transformers that implement building blocks for classic analysis parameters like context, path, and heap (in)sensitivity. Moreover, these can be composed together *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 transformer stack is sound by virtue of its constituents. Galois transformers provide a viable foundation for reusable 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 program analysis via abstract interpretation is to fix a language (as a concrete semantics) and an abstraction (as an abstraction map, concretization map or Galois connection) before constructing a static analyzer that is sound with respect to both the abstraction and the concrete semantics. Thus, each pairing of abstraction and semantics requires a one-off manual derivation of the abstract semantics and construction of a proof of soundness.

Work has focused on endowing abstractions with knobs, levers, and dials to tune precision and compute efficiently. These parameters come 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 this framework approach suffers from many of the same drawbacks as the one-off analyzers. They are language-specific, preventing reuse of concepts across languages, and require similar re-implementations and soundness proofs. This process is still manual, tedious, difficult and error-prone. And, changes to the structure of the parameter-space require a completely new proof of soundness. And, it prevents fruitful insights and results developed in one paradigm from being applied to others, e.g., functional to object-oriented and *vice versa*.

We propose an automated alternative to structuring and implementing program analysis. Inspired by Liang, Hudak, and Jones’s *Monad Transformers and Modular Interpreters* [11], we propose to start with concrete interpreters written in a specific monadic style. Changing the monad will change the interpreter from a concrete interpreter into an abstract interpreter. 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 transformers obey the properties of *Galois connections* [5] and introduce the concept of a *Galois transformer*, a monad transformer which transports (1) Galois connections and (2) mappings to an executable transition system.

Most significantly, Galois transformers can be proved sound once and used everywhere. Abstract interpreters, which take the form of monad transformer stacks coupled 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.

Setup We describe a simple language and a garbage-collecting allocating semantics as the starting point of analysis design (Section 2). We then briefly discuss three types of path and flow sensitivity and their corresponding variations in analysis precision (Section 3).

Monadic Abstract Interpreters We develop an abstract interpreter for our example language as a monadic function with parameters (Section 4), one of which is a monadic effect interface combining state and nondeterminism effects (Section 4.1). These monadic effects, state and nondeterminism, encode arbitrary relational small-step state-machine semantics and correspond to state-machine components and relational nondeterminism respectively.

Interpreters written in this style can be reasoned about using various laws, including monadic effect laws, and therefore verified correct independent of any particular choice of parameters. Likewise, instantiations for these parameters can be reasoned about in isolation from their instantiation. When instantiated, our generic interpreter is capable of recovering the concrete semantics and a family of abstract interpreters, with variations in abstract domain, abstract garbage collection, mcfa, call-site sensitivity, object sensitivity, and path and flow sensitivity (Section 6).

Isolating Path and Flow Sensitivity We give specific monads for instantiating the interpreter from Section 5 to path-sensitive, flow-sensitive and flow-insensitive analyses (Section 7). This leads to an isolated understanding of path and flow sensitivity as mere variations in the monad used for execution. Furthermore, these monads are language independent, allowing one to reuse the same path and flow sensitivity machinery for any language of interest, and compose seamlessly with other analysis parameters.

Galois Transformers To ease the construction of monads for building abstract interpreters and their proofs of correctness, we develop a framework of Galois transformers (Section 8). Galois transformers are an extension of monad transformers which transport (1) Galois connections and (2) mappings to an executable transition system (Section 8.5). Our Galois transformer framework allows us to both execute and reason about the correctness of an abstract interpreter piecewise for each transformer in a stack. Galois transformers are language independent and they can be proven correct one and for all in isolation from a particular semantics.

Implementation We have implemented our technique as a Haskell library and example client analysis (Section 9). Developers are able to reuse our language-independent framework for prototyping the design space of analysis features for their language of choice. Our implementation is publicly available on Hackage¹, Haskell’s package manager.

Contributions We make the following contributions:

- A methodology for constructing monadic abstract interpreters based on *monadic effects*².

¹ <http://hackage.haskell.org/package/maam>

² This is in contrast to Sergey et al. [18] where monadic interpreters are constructed based on *denotation functions*. See our Section 10 for more details.

$i \in \mathbb{Z}$	
$x \in Var$	
$a \in Atom$	$::= i \mid x \mid \underline{\lambda}(x).e$
$\oplus \in IOp$	$::= + \mid -$
$\odot \in Op$	$::= \oplus \mid @$
$e \in Exp$	$::= a \mid e \odot e \mid \text{if0}(e)\{e\}\{e\}$
$\tau \in Time$	$::= \mathbb{Z}$
$l \in Addr$	$::= Var \times Time$
$\rho \in Env$	$::= Var \rightarrow Addr$
$\sigma \in Store$	$::= Addr \rightarrow Val$
$c \in Clo$	$::= \langle \underline{\lambda}(x).e, \rho \rangle$
$v \in Val$	$::= i \mid c$
$\kappa l \in KAddr$	$::= Time$
$\kappa \sigma \in KStore$	$::= KAddr \rightarrow Frame \times KAddr$
$fr \in Frame$	$::= \langle \square \odot e \rangle \mid \langle v \odot \square \rangle \mid \langle \text{if0}(\square)\{e\}\{e\} \rangle$
$\varsigma \in \Sigma$	$::= Exp \times Env \times Store \times KAddr \times KStore$

Figure 1: λIF Syntax and Concrete State Space

- A compositional, language-independent library for constructing monads with various analysis properties based on *monad transformers*.
- A compositional, language-independent proof framework for constructing Galois connections using *Galois transformers*, an extension of monad transformers which transport (1) Galois connections and (2) mappings to an executable transition system.
- Two new general purpose monad transformers for nondeterminism which are not present in any previous work on monad transformers (even outside static analysis literature). Although applicable to settings other than static analysis, these two transformers give rise naturally to variations in path and flow sensitivity when applied to abstract interpreters.
- An isolated understanding of path and flow sensitivity in analysis as properties of the interpreter monad, which we develop independently of other analysis features.

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 write $@$ as explicit abstract syntax for function application. The state-space Σ for λIF makes allocation explicit using two separate stores for values (*Store*) and for the stack (*KStore*).

Guided by the syntax and semantics of λIF defined in this section we develop interpretation parameters in Section 4, a monadic interpreter in Section 5, and both concrete

$$\begin{aligned}
A[_] &\in Atom \rightarrow (Env \times Store \rightarrow Val) \\
A[i](\rho, \sigma) &:= i \\
A[x](\rho, \sigma) &:= \sigma(\rho(x)) \\
A[\lambda(x).e](\rho, \sigma) &:= \langle \lambda(x).e, \rho \rangle \\
\delta[_] &\in IOp \rightarrow (\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}) \\
\delta[+](i_1, i_2) &:= i_1 + i_2 \\
\delta[-](i_1, i_2) &:= i_1 - i_2 \\
_ \rightsquigarrow _ &\in \mathcal{P}(\Sigma \times \Sigma) \\
\langle e_1 \odot e_2, \rho, \sigma, \kappa l, \kappa \sigma, \tau \rangle &\rightsquigarrow \langle e_1, \rho, \sigma, \tau, \kappa \sigma', \tau + 1 \rangle \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 &\rightsquigarrow \langle e, \rho, \sigma, \tau, \kappa \sigma', \tau + 1 \rangle \text{ where} \\
&\langle \langle \square \odot e \rangle, \kappa l' \rangle := \kappa \sigma(\kappa l) \\
&\kappa \sigma' := \kappa \sigma[\tau \mapsto (\langle A[a](\rho, \sigma) \odot \square \rangle, \kappa l')] \\
\langle a, \rho, \sigma, \kappa l, \kappa \sigma, \tau \rangle &\rightsquigarrow \langle e, \rho'', \sigma', \kappa l', \kappa \sigma, \tau + 1 \rangle \text{ where} \\
&\langle \langle \lambda(x).e, \rho' \rangle @ \square, \kappa l' \rangle := \kappa \sigma(\kappa l) \\
&\rho'' := \rho'[x \mapsto (x, \tau)] \\
&\sigma' := \sigma[(x, \tau) \mapsto A[a](\rho, \sigma)] \\
\langle i_2, \rho, \sigma, \kappa l, \kappa \sigma, \tau \rangle &\rightsquigarrow \langle i, \rho, \sigma, \kappa l', \kappa \sigma, \tau + 1 \rangle \text{ where} \\
&\langle \langle i_1 \oplus \square \rangle, \kappa l' \rangle := \kappa \sigma(\kappa l) \\
&i := \delta[+](i_1, i_2) \\
\langle i, \rho, \sigma, \kappa l, \kappa \sigma, \tau \rangle &\rightsquigarrow \langle e, \rho, \sigma, \kappa l', \kappa \sigma, \tau + 1 \rangle \text{ where} \\
&\langle \langle \text{if0}(\square)\{e_1\}\{e_2\} \rangle, \kappa l' \rangle := \kappa \sigma(\kappa l) \\
&e := e_1 \text{ when } i = 0; e_2 \text{ when } i \neq 0
\end{aligned}$$

Figure 2: Concrete Semantics

and abstract instantiations for the interpretation parameters in Section 6. The variations in path and flow sensitivity developed in sections 7 and 8 are independent of this (or any other) semantics.

We give semantics to atomic expressions and primitive operators denotationally through $A[_]$ and $\delta[_]$, and to compound expressions relationally through \rightsquigarrow , as shown in Figure 2. We will recover these semantics from a concrete instantiation of our generic abstract interpreter in Section 6.

Our abstract interpreter will support abstract garbage collection [13], the concrete analogue of which is just standard garbage collection. We include abstract 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 systematically recover both concrete and abstract garbage collectors through a single monadic garbage collector.

Garbage collection is defined using a reachability function R which computes transitively reachable address in σ :

$$\begin{aligned}
R &\in Store \times Env \times Exp \rightarrow \mathcal{P}(Addr) \\
R(\sigma, \rho, e) &:= \mu(X). \\
&X \cup R_0(\rho, e) \cup \{l' \mid l' \in R\text{-}Val(\sigma(l)); l \in X\}
\end{aligned}$$

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 $R\text{-}Val$ for computing addresses reachable from values.

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

We omit the definition of FV , which is the standard recursive definition for computing free variables of an expression. Analogously, KR is the set of transitively reachable stack-frame addresses in $\kappa \sigma$:

$$\begin{aligned}
KR &\in KStore \times KAddr \rightarrow \mathcal{P}(KAddr) \\
KR(\kappa \sigma, \kappa l_0) &:= \mu(X).X \cup \{\kappa l_0\} \cup \{\pi_2(\kappa \sigma(\kappa l)) \mid \kappa l \in X\}
\end{aligned}$$

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

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

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

$$Analysis := \mu(X).X \cup \{\varsigma_0\} \cup \{\varsigma' \mid \varsigma \rightsquigarrow^{gc} \varsigma'; \varsigma \in X\}$$

where ς_0 is the injection of the initial program e_0 :

$$\varsigma_0 := \langle e_0, \perp, \perp, 0, \perp, 1 \rangle$$

The analyses we present in this paper will be proven correct in Section 6 by establishing a Galois connection with this concrete collecting semantics.

3. Path and Flow Sensitivity in Analysis

In this paper we identify three specific variants of path and flow sensitivity in analysis: path-sensitive, flow-sensitive and flow-insensitive. Our framework exposes the essence of path and flow sensitivity through a monadic effect interface in Section 4, and we recover each of these variations through specific monad instances in Sections 7 and 8.

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

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

Path-Sensitive A path-sensitive analysis tracks both data and control flow precisely. At program points 3 and 4 the

analysis considers separate worlds:

$$3 : \{N = 0\} \quad 4 : \{N \neq 0\}$$

At program points 5 and 6 the analysis continues in two separate, precise worlds:

$$5, 6 : \{N = 0, x = 1\} \{N \neq 0, x = 4\}$$

At program point 7 the analysis correctly correlates the values for x and y :

$$7 : \{N = 0, x = 1, y = 5\} \{N \neq 0, x = 4, y = 6\}$$

Flow-Sensitive A flow-sensitive analysis collects a *single* set of facts for each variable *at each program point*. At program points 3 and 4, the analysis considers separate worlds:

$$3 : \{N = 0\} \quad 4 : \{N \neq 0\}$$

Each nested if-statement then evaluates only one side of the branch, resulting in values 1 and 4. At program points 5 and 6 the analysis is only allowed one set of facts, so it must merge the possible values that x and N could take:

$$5, 6 : \{N \in \mathbb{Z}, x \in \{1, 4\}\}$$

The analysis then explores both branches at program point 6 resulting in no correlation between values for x and y :

$$7 : \{N \in \mathbb{Z}, x \in \{1, 4\}, y \in \{5, 6\}\}$$

Flow-Insensitive A flow-insensitive analysis collects a *single* set of facts about each variable which must hold true *for the entire program*. Because the value of N is unknown at *some* point in the program, the value of x must consider both branches of the nested if-statement. This results in the global set of facts giving four values to x .

$$\{N \in \mathbb{Z}, x \in \{1, 2, 3, 4\}, y \in \{5, 6\}\}$$

In our framework we capture the variation of path and flow sensitivity as a purely orthogonal parameter to the abstract interpreter. Path and flow sensitivity will then compose seamlessly with choices of call-site sensitivity, object sensitivity, abstract garbage collection, mcfa a la Might et al. [14], shape analysis, abstract domain, etc. Most importantly, we empower the analysis designer to compartmentalize the path and flow sensitivity of each component in the abstract state space independently. Constructing an analysis which is flow-sensitive in the data-store and path-sensitive in the stack-store is just as easy as constructing a single path or flow sensitivity across the board, and one can alternate between them for free.

4. Analysis Parameters

Before writing an abstract interpreter we first design its parameters. The interpreter, which we develop in Section 5, will be designed such that variations in these parameters will recover both concrete and a family of abstract interpreters, which we show in Section 6. To do this we extend the ideas developed in Van Horn and Might [22] with a new parameter for path and flow sensitivity.

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

1. The monad, novel in this work, captures the state and control effects of the interpreter and gives rise to path and flow sensitivity.
2. The abstract domain, which for this language is merely an abstraction for integers.
3. Abstract Time, capturing call-site and object sensitivities.

We place each of these parameters behind an abstract interface and leave their implementations opaque when defining the monadic interpreter in Section 5. Each of these parameters come with reasoning principles. These principles allow us to reason about the correctness of the generic interpreter independent of a particular instantiation of the parameters. The goal is to factor as much of the proof effort into what we can say about the generic interpreter independent of parameter instantiation. Then, an instantiation of the interpreter need only justify that each parameter meets its local interface.

4.1 The Analysis Monad

The monad for the interpreter captures the *effects* of interpretation. There are two effects in the interpreter: state and nondeterminism. The state effect will mediate how the interpreter interacts with state cells in the state space: *Env*, *Store*, *KAddr* and *KStore*. The nondeterminism effect will mediate branching in the execution of the interpreter. One of our results is that path and flow sensitivity can be recovered by altering how these effects interact in the monad.

We use monadic state and nondeterminism effects to abstract over arbitrary *relational small-step state-machine semantics*. State effects correspond to the components of the state-machine and nondeterminism effects correspond to potential nondeterminism in the relation's definition.

We briefly review monad, state and nondeterminism operators and their laws. For more details about monad transformers and monad laws we refer the reader to [11].

Monadic Sequencing A type operator M is a monad if it supports *bind*, a sequencing operator, and its unit *return*.

$$M : Type \rightarrow Type$$

$$bind : \forall \alpha \beta, M(\alpha) \rightarrow (\alpha \rightarrow M(\beta)) \rightarrow M(\beta)$$

$$return : \forall \alpha, \alpha \rightarrow M(\alpha)$$

We use monad laws (left and right units, and associativity) to reason about the interpreter in the absence of a particular implementation of *bind* and *return*. We use semicolon notation as syntactic sugar for *bind*; for example, $a \leftarrow m ; k(a)$ is sugar for $bind(m)(k)$. We replace semicolons with line breaks headed by a **do** command for multiline monadic definitions.

State Effect A type operator M supports the monadic state effect for a type s if it supports *get* and *put* actions over s .

$$M : Type \rightarrow Type \quad get : M(s)$$

$$s : Type \quad put : s \rightarrow M(1)$$

We use the state monad laws (get-get, get-put, put-get, put-put) to reason about state effects.

Nondeterminism Effect A type operator M supports the monadic nondeterminism effect if it supports an alternation operator $\langle + \rangle$ and its unit $mzero$.

$$\begin{aligned} M &: Type \rightarrow Type \\ _ \langle + \rangle _ &: \forall \alpha, M(\alpha) \times M(\alpha) \rightarrow M(\alpha) \\ mzero &: \forall \alpha, M(\alpha) \end{aligned}$$

Nondeterminism laws state that $M(\alpha)$ must have a join-semilattice structure, that $mzero$ be a zero for $bind$, and that $bind$ distributes through $\langle + \rangle$.

Monad Examples The state monad over s is defined with $s \rightarrow (\alpha \times s)$ and supports monadic sequencing ($bind$ and $return$) and state effects (get and put). The nondeterminism monad is defined with $\mathcal{P}(\alpha)$ and supports monadic sequencing ($bind$ and $return$) and nondeterminism effects ($_ \langle + \rangle _$ and $mzero$).

Our interpreter will be defined up to this effect interface and will avoid referencing an explicit configuration ς or collections of results. This level of indirection will then be exploited in Section 7, where different monads will meet the same effect interface but yield different analysis properties.

4.2 The Abstract Domain

The abstract domain is the Val type in the semantics. To parameterize over the abstract domain we make Val opaque, but require that it support various operations.

Val must be a join-semilattice with \perp and \sqcup respecting the usual laws. We require Val to be a join-semilattice so it can be merged in updates to $Store$ to preserve soundness.

$$\perp : Val \quad _ \sqcup _ : Val \times Val \rightarrow Val$$

Val must also support conversions to and from concrete values. These conversions take the form of introduction and elimination rules. Introduction rules inject concrete values into abstract values. Elimination rules project abstract values into a *finite* set of concrete observations. For example, we do not require that abstract values support elimination to integers, only the finite observation of comparison with zero.

$$\begin{aligned} int-I &: \mathbb{Z} \rightarrow Val & int-if0-E &: Val \rightarrow \mathcal{P}(Bool) \\ clo-I &: Clo \rightarrow Val & clo-E &: Val \rightarrow \mathcal{P}(Clo) \end{aligned}$$

The laws for the introduction and elimination rules induce a Galois connection between $\mathcal{P}(\mathbb{Z})$ and Val :

$$\begin{aligned} \{\mathbf{true}\} &\sqsubseteq int-if0-E(int-I(i)) \text{ if } i = 0 \\ \{\mathbf{false}\} &\sqsubseteq int-if0-E(int-I(i)) \text{ if } i \neq 0 \\ \bigsqcup_{\substack{b \in int-if0-E(v) \\ i \in \theta(b)}} int-I(i) &\sqsubseteq v \\ \text{where } \theta(\mathbf{true}) &:= \{0\} \\ \theta(\mathbf{false}) &:= \{i \mid i \in \mathbb{Z}; i \neq 0\} \end{aligned}$$

Closures must follow similar laws, inducing a Galois connection between $\mathcal{P}(Clo)$ and Val :

$$\begin{aligned} \{c\} &\sqsubseteq clo-E(cloI(c)) \\ \bigsqcup_{c \in clo-E(v)} clo-I(c) &\sqsubseteq v \end{aligned}$$

Finally, δ must be sound and complete w.r.t. the Galois connection between concrete values and Val :

$$\begin{aligned} int-I(i_1 + i_2) &\sqsubseteq \delta[\![+]\!](int-I(i_1), int-I(i_2)) \\ int-I(i_1 - i_2) &\sqsubseteq \delta[\![-]\!](int-I(i_1), int-I(i_2)) \\ \bigsqcup_{\substack{b_1 \in int-if0-E(v_1) \\ b_2 \in int-if0-E(v_2) \\ i \in \theta(b_1, b_2)}} int-I(i) &\sqsubseteq \delta[\![\odot]\!](v_1, v_2) \\ \theta(\mathbf{true}, \mathbf{true}) &:= \{0\} \\ \theta(\mathbf{true}, \mathbf{false}) &:= \{i \mid i \in \mathbb{Z}; i \neq 0\} \\ \text{where } \theta(\mathbf{false}, \mathbf{true}) &:= \{i \mid i \in \mathbb{Z}; i \neq 0\} \\ \theta(\mathbf{false}, \mathbf{false}) &:= \mathbb{Z} \end{aligned}$$

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

4.3 Abstract Time

The interface for abstract time is familiar from Van Horn and Might [22] (AAM), which introduces abstract time as a single parameter to control various forms of context sensitivity, and Smaragdakis et al. [21], which instantiates the parameter to achieve both call-site and object sensitivity. We only demonstrate call-site sensitivity in this presentation, but our language-independent library supports object sensitivity, the implementation of which is a straightforward application of Smaragdakis et al. [21].

$$Time : Type \quad tick : Exp \times KAddr \times Time \rightarrow Time$$

Remarkably, we need not state laws for $tick$. The interpreter will merge values which reside at the same address to preserve soundness. Therefore, any supplied implementations of $tick$ is valid from a soundness perspective. However, different choices in $tick$ will yield different trade-offs in precision and performance of the abstract interpreter.

5. The Interpreter

We now present a monadic interpreter for λIF parameterized over M , Val and $Time$ from Section 4. We instantiate these parameters to obtain an analysis in Section 6.

First we implement $A[_]$ as a *monadic* denotation for atomic expressions, shown in Figure 3. The monadic $A[_]$ is a straightforward translation of the $A[_]$ shown in Figure 2 from a pure function to a monadic function with state effects. $get-Env$ and $get-Store$ are primitive operations for monadic state. $clo-I$ comes from the interface for Val .

Next we implement *step*, a *monadic small-step function* for compound expressions, also shown in Figure 3. The monadic *step* is a straightforward translation of the *step* shown in Figure 2 from a relation to a monadic function with state and nondeterminism effects.

step uses helper functions *push* and *pop* for manipulating stack frames, \uparrow_p for lifting values from \mathcal{P} into M , and a monadic version of *tick* called *tickM*, each of which are shown in Figure 4. The interpreter looks deterministic, however the nondeterminism is abstracted away behind \uparrow_p and monadic bind $x \leftarrow e_1 ; e_2$.

We implement abstract garbage collection in a general away using the monadic effect interface, also shown in Figure 3. R and KR are as defined in Section 2. Again, this is a straightforward translation from a pure function to a monadic function with state effects.

Preserving Soundness In generalizing the semantics to account for nondeterminism, updates to both the data-store and stack-store must merge values rather than performing a strong update. This is because we place no restriction on the semantics for *Time* and therefore must preserve soundness in the presence of reused addresses.

To support \sqcup for stores (in observation of soundness) we modify our definitions of *Store* and *KStore*.

$$\begin{aligned}\sigma &\in \text{Store} : \text{Addr} \rightarrow \text{Val} \\ \kappa\sigma &\in \text{KStore} : \text{KAddr} \rightarrow \mathcal{P}(\text{Frame} \times \text{KAddr})\end{aligned}$$

Execution In the concrete semantics, execution takes the form of a least-fixed-point computation over a collecting semantics. This in general requires a join-semilattice structure for some Σ and a transition function $\Sigma \rightarrow \Sigma$. However, we no longer have a function $\Sigma \rightarrow \Sigma$; we have a monadic function $\text{Exp} \rightarrow M(\text{Exp})$ which does not immediately admit a least-fixed-point iteration to execute the analysis.

To solve this we require that monadic actions $\text{Exp} \rightarrow M(\text{Exp})$ 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. This will allow us to construct Galois connections between monads $m_1 \xrightarrow[\alpha]{\gamma} m_2$ and derive Galois connections between transition systems $\Sigma_1 \xrightarrow[\alpha]{\gamma} \Sigma_2$ for free.

A collecting-semantics execution of our interpreter is then defined as the least-fixed-point iteration of *step* transported through the Galois connection:

$$\text{Analysis} := \mu(X). X \sqcup \varsigma_0 \sqcup \gamma(\text{step})(X)$$

where ς_0 is the injection of the initial program e_0 into Σ and γ has type $(\text{Exp} \rightarrow M(\text{Exp})) \rightarrow (\Sigma \rightarrow \Sigma)$.

```

 $A[\_]\in \text{Atom} \rightarrow M(\text{Val})$ 
 $A[i] := \text{return}(\text{int-}I(i))$ 
 $A[x] := \text{do}$ 
   $\rho \leftarrow \text{get-Env}$ 
   $\sigma \leftarrow \text{get-Store}$ 
  if  $x \in \rho$  then  $\text{return}(\sigma(\rho(x)))$  else  $\text{return}(\perp)$ 
 $A[\underline{\lambda}(x).e] := \text{do}$ 
   $\rho \leftarrow \text{get-Env}$ 
   $\text{return}(\text{clo-}I(\langle \underline{\lambda}(x).e, \rho \rangle))$ 
 $\text{step} : \text{Exp} \rightarrow M(\text{Exp})$ 
 $\text{step}(e) := \text{do}$ 
   $\text{tickM}(e)$ 
   $e' \leftarrow \text{case } e \text{ of}$ 
     $e_1 \odot e_2 \rightarrow \text{do}$ 
       $\text{push}(\langle \square \odot e_2 \rangle)$ 
       $\text{return}(e_1)$ 
     $a \rightarrow \text{do}$ 
       $fr \leftarrow \text{pop}$ 
       $v \leftarrow A[a]$ 
       $e \leftarrow \text{case } fr \text{ of}$ 
         $\langle \square \odot e \rangle \rightarrow \text{do}$ 
           $\text{push}(\langle v \odot \square \rangle)$ 
           $\text{return}(e)$ 
         $\langle v' @ \square \rangle \rightarrow \text{do}$ 
           $\langle \underline{\lambda}(x).e, \rho' \rangle \leftarrow \uparrow_p(\text{clo-}E(v'))$ 
           $\tau \leftarrow \text{get-Time}$ 
           $\sigma \leftarrow \text{get-Store}$ 
           $\text{put-Env}(\rho'[x \mapsto (x, \tau)])$ 
           $\text{put-Store}(\sigma \sqcup [(x, \tau) \mapsto \{v\}])$ 
           $\text{return}(e)$ 
         $\langle v' \oplus \square \rangle \rightarrow \text{do}$ 
           $\text{return}(\delta[\oplus](v', v))$ 
       $\langle \text{if0}(\square)\{e_1\}\{e_2\} \rangle \rightarrow \text{do}$ 
         $b \leftarrow \uparrow_p(\text{int-if0-}E(v))$ 
        if  $(b)$  then  $\text{return}(e_1)$  else  $\text{return}(e_2)$ 
   $gc(e')$ 
   $\text{return}(e')$ 
 $gc : \text{Exp} \rightarrow M(1)$ 
 $gc(e) := \text{do}$ 
   $\rho \leftarrow \text{get-Env}$ 
   $\sigma \leftarrow \text{get-Store}$ 
   $\kappa\sigma \leftarrow \text{get-KStore}$ 
   $\text{put-Store}(\{l \mapsto \sigma(l) \mid l \in R(\sigma, \rho, e)\})$ 
   $\text{put-KStore}(\{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in KR(\kappa\sigma, \kappa l)\})$ 

```

```

 $\uparrow_p : \forall \alpha, \mathcal{P}(\alpha) \rightarrow M(\alpha)$ 
 $\uparrow_p(\{a_1..a_n\}) := \text{return}(a_1) \langle + \rangle .. \langle + \rangle \text{return}(a_n)$ 
 $\text{push} : \text{Frame} \rightarrow M(1)$ 
 $\text{push}(fr) := \text{do}$ 
   $\kappa l \leftarrow \text{get-KAddr}$ 
   $\kappa \sigma \leftarrow \text{get-KStore}$ 
   $\kappa l' \leftarrow \text{get-Time}$ 
   $\text{put-KStore}(\kappa \sigma \sqcup [\kappa l' \mapsto \{fr :: \kappa l\}])$ 
   $\text{put-KAddr}(\kappa l')$ 
 $\text{pop} : M(\text{Frame})$ 
 $\text{pop} := \text{do}$ 
   $\kappa l \leftarrow \text{get-KAddr}$ 
   $\kappa \sigma \leftarrow \text{get-KStore}$ 
   $fr :: \kappa l' \leftarrow \uparrow_p(\kappa \sigma(\kappa l))$ 
   $\text{put-KAddr}(\kappa l')$ 
   $\text{return}(fr)$ 
 $\text{tickM} : \text{Exp} \rightarrow M(1)$ 
 $\text{tickM}(e) = \text{do}$ 
   $\tau \leftarrow \text{get-Time}$ 
   $\kappa l \leftarrow \text{get-KAddr}$ 
   $\text{put-Time}(\text{tick}(e, \kappa l, \tau))$ 

```

Figure 4: Monadic helper functions

6. Recovering Analyses

In Section 5 we defined a monadic interpreter with the uninstantiated parameters from Section 4: M , Val and $Time$. To recover a concrete interpreter we instantiate these parameters to concrete components \mathbf{M} , \mathbf{Val} and \mathbf{Time} , and to recover an abstract interpreter we instantiate them to abstract components $\widehat{\mathbf{M}}$, $\widehat{\mathbf{Val}}$ and $\widehat{\mathbf{Time}}$. The soundness of the final implementation is thus factored into two steps:

1. Proving the parameterized monadic interpreter correct for any instantiation of M , Val and $Time$.
2. Constructing Galois connections $\mathbf{M} \xleftarrow[\alpha]{\gamma} \widehat{\mathbf{M}}$, $\mathbf{Val} \xleftarrow[\alpha]{\gamma} \widehat{\mathbf{Val}}$ and $\mathbf{Time} \xleftarrow[\alpha]{\gamma} \widehat{\mathbf{Time}}$ piecewise.

The key benefit of our approach is that (1) can be proved once against *all* instantiations of M , Val and $Time$ using the reasoning principles established in Section 4, greatly simplifying the proof burden when choosing different abstract components in (2).

6.1 Recovering a Concrete Interpreter

For the concrete value space we instantiate Val to \mathbf{Val} :

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

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

$$\text{int-I} : \mathbb{Z} \rightarrow \mathbf{Val}$$

$$\text{int-I}(i) := \{i\}$$

$$\text{int-if0-E} : \mathbf{Val} \rightarrow \mathcal{P}(\text{Bool})$$

$$\text{int-if0-E}(v) := \{\text{true} \mid 0 \in v\} \cup \{\text{false} \mid \exists i \in v.s.t.i \neq 0\}$$

and a straightforward concrete δ :

$$\delta[_]_ : \text{IOp} \rightarrow \mathbf{Val} \times \mathbf{Val} \rightarrow \mathbf{Val}$$

$$\delta[_+](v_1, v_2) := \{i_1 + i_2 \mid i_1 \in v_1 ; i_2 \in v_2\}$$

$$\delta[_ -](v_1, v_2) := \{i_1 - i_2 \mid i_1 \in v_1 ; i_2 \in v_2\}$$

Proposition 1. *Val satisfies the abstract domain laws shown in Section 4.2.*

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

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

and tick is just a cons operator:

$$\text{tick} : \text{Exp} \times \mathbf{KAddr} \times \mathbf{Time} \rightarrow \mathbf{Time}$$

$$\text{tick}(e, \kappa l, \tau) := (e, \kappa l) :: \tau$$

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:

$$\text{bind} : \forall \alpha, \mathbf{M}(\alpha) \rightarrow (\alpha \rightarrow \mathbf{M}(\beta)) \rightarrow \mathbf{M}(\beta)$$

$$\text{bind}(m)(f)(\psi) :=$$

$$\{(y, \psi'') \mid (y, \psi') \in f(a)(\psi') ; (a, \psi') \in m(\psi)\}$$

$$\text{return} : \forall \alpha, \alpha \rightarrow \mathbf{M}(\alpha)$$

$$\text{return}(a)(\psi) := \{(a, \psi)\}$$

State effects return singleton sets:

$$\text{get-Env} : \mathbf{M}(\mathbf{Env})$$

$$\text{get-Env}(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(\rho, \langle \rho, \sigma, \kappa, \tau \rangle)\}$$

$$\text{put-Env} : \mathbf{Env} \rightarrow \mathbf{P}(1)$$

$$\text{put-Env}(\rho')(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(1, \langle \rho', \sigma, \kappa, \tau \rangle)\}$$

Nondeterminism effects are implemented with set union:

$$\text{mzero} : \forall \alpha, \mathbf{M}(\alpha)$$

$$\text{mzero}(\psi) := \{\}$$

$$_ \langle + \rangle _ : \forall \alpha, \mathbf{M}(\alpha) \times \mathbf{M}(\alpha) \rightarrow \mathbf{M}(\alpha)$$

$$(m_1 \langle + \rangle m_2)(\psi) := m_1(\psi) \cup m_2(\psi)$$

Proposition 2. *M satisfies monad, state, and nondeterminism laws shown in Section 4.1.*

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

$$\Sigma := \mathcal{P}(\text{Exp} \times \Psi)$$

The Galois connection between \mathbf{M} and Σ is similar to the definition of *bind*:

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

The injection ς_0 for a program e_0 is $\{\langle e_0, \perp, \perp, \bullet, \perp, \bullet \rangle\}$.

Proposition 3. 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 abstract value domains.

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

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

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

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

The abstract δ operator is defined:

$$\begin{aligned}\delta &: IOp \rightarrow \widehat{\mathbf{Val}} \times \widehat{\mathbf{Val}} \rightarrow \widehat{\mathbf{Val}} \\ \delta[\![+]\!](v_1, v_2) &:= \\ &\quad \{i \mid 0 \in v_1 \wedge i \in v_2\} \cup \{i \mid i \in v_1 \wedge 0 \in v_2\} \\ &\quad \cup \{+ \mid + \in v_1 \wedge + \in v_2\} \cup \{- \mid - \in v_1 \wedge - \in v_2\} \\ &\quad \cup \{-, 0, + \mid + \in v_1 \wedge - \in v_2\} \\ &\quad \cup \{-, 0, + \mid - \in v_1 \wedge + \in v_2\}\end{aligned}$$

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.

Proposition 5. $\mathbf{Val} \xleftrightarrow[\alpha]{\gamma} \widehat{\mathbf{Val}}$, and both concrete and abstract definitions for *int-I*, *int-if0-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:

$$\widehat{\mathbf{Time}} := (Exp \times \widehat{\mathbf{KAddr}})^{*k}$$

The *tick* operator becomes cons followed by k-truncation, which restricts the list to the first-k elements:

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

This abstraction for time yields k-call-site sensitivity, or a kCFA analysis.

Proposition 6. $\mathbf{Time} \xleftrightarrow[\alpha]{\gamma} \widehat{\mathbf{Time}}$, and both concrete and abstract definitions for *tick* are 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 .

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

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

7. Varying Path and Flow Sensitivity

Sections 5 and 6 describe the construction of a path-sensitive analysis using our framework. In this section we show an alternate definition for $\widehat{\mathbf{M}}$ which yields a flow-insensitive analysis. Section 8 will generalize the definitions from this section into compositional components (monad transformers) in addition to introducing another definition for $\widehat{\mathbf{M}}$ which yields a flow-sensitive analysis.

Before going into the details of the flow-insensitive monad, we wish to build intuition regarding what one would expect from such a development.

Recall the path-sensitive monad $\widehat{\mathbf{M}}$ and its state space $\widehat{\Sigma}$ from section 6:

$$\begin{aligned}\widehat{\mathbf{M}}(Exp) &:= \Psi \times \widehat{\mathbf{Store}} \rightarrow \mathcal{P}(Exp \times \Psi \times \widehat{\mathbf{Store}}) \\ \widehat{\Sigma}(Exp) &:= \mathcal{P}(Exp \times \Psi \times \widehat{\mathbf{Store}})\end{aligned}$$

where $\Psi := \widehat{\mathbf{Env}} \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{KStore}} \times \widehat{\mathbf{Time}}$. This is path-sensitive because $\widehat{\Sigma}(Exp)$ can represent arbitrary relations between $(Exp \times \Psi)$ and $\widehat{\mathbf{Store}}$.

As discussed in Section 3, a flow-insensitive analysis will give a single set of facts per program point. This results in the following monad $\widehat{\mathbf{M}}^{fs}$ and state space $\widehat{\Sigma}^{fs}$ which encode finite maps to $\widehat{\mathbf{Store}}$ rather than relations:

$$\begin{aligned}\widehat{\mathbf{M}}^{fs}(Exp) &:= \Psi \times \widehat{\mathbf{Store}} \rightarrow [(Exp \times \Psi) \mapsto \widehat{\mathbf{Store}}] \\ \widehat{\Sigma}^{fs}(Exp) &:= [(Exp \times \Psi) \mapsto \widehat{\mathbf{Store}}]\end{aligned}$$

where $[\alpha \mapsto s]$ is notation for a finite map from α to s .

Finally, a flow-insensitive analysis contains a single global set of facts, which is represented by pulling $\widehat{\mathbf{Store}}$ out of the powerset:

$$\begin{aligned}\widehat{\mathbf{M}}^{fi}(Exp) &:= \Psi \times \widehat{\mathbf{Store}} \rightarrow \mathcal{P}(Exp \times \Psi) \times \widehat{\mathbf{Store}} \\ \widehat{\Sigma}^{fi}(Exp) &:= \mathcal{P}(Exp \times \Psi) \times \widehat{\mathbf{Store}}\end{aligned}$$

These three resulting structures, $\widehat{\Sigma}$, $\widehat{\Sigma}^{fs}$ and $\widehat{\Sigma}^{fi}$, capture the essence of path-sensitive, flow-sensitive and flow-insensitive iteration, and arise naturally from $\widehat{\mathbf{M}}$, $\widehat{\mathbf{M}}^{fs}$ and $\widehat{\mathbf{M}}^{fi}$, which each have monadic structure. We only describe $\widehat{\mathbf{M}}^{fi}$ directly in this section; more compositional versions of each of these monads are described in Section 8, from which $\widehat{\Sigma}^{fs}$ and $\widehat{\Sigma}^{fi}$ can be recovered.

7.1 Flow Insensitive Monad

For $\widehat{\mathbf{M}}^{fi}$ the monad operator *bind* performs the store merging needed to capture a flow-insensitive analysis.

$$\begin{aligned} bind : \forall \alpha \beta, \widehat{\mathbf{M}}^{fi}(\alpha) &\rightarrow (\alpha \rightarrow \widehat{\mathbf{M}}^{fi}(\beta)) \rightarrow \widehat{\mathbf{M}}^{fi}(\beta) \\ bind(m)(f)(\psi, \sigma) &:= \\ &(\{bs_{11}..bs_{1m_1}..bs_{n1}..bs_{nm_n}\}, \sigma_1 \sqcup .. \sqcup \sigma_n) \text{ where} \\ &(\{(a_1, \psi_1)..(a_n, \psi_n)\}, \sigma') := m(\psi, \sigma) \\ &(\{b\psi_{i1}..b\psi_{im_i}\}, \sigma_i) := f(a_i)(\psi_i, \sigma') \end{aligned}$$

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

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

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

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

State effects *get-Store* and *put-Store* are analogous to *get-Env* and *put-Env*.

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

$$\begin{aligned} mzero : \forall \alpha, M(\alpha) \\ mzero(\psi, \sigma) &:= (\{\}, \perp) \\ _ \langle + \rangle _ : \forall \alpha, M(\alpha) \times M(\alpha) &\rightarrow M(\alpha) \\ (m_1 \langle + \rangle m_2)(\psi, \sigma) &:= (\alpha\psi_1^* \cup \alpha\psi_2^*, \sigma_1 \sqcup \sigma_2) \text{ where} \\ (\alpha\psi_i^*, \sigma_i) &:= m_i(\psi, \sigma) \end{aligned}$$

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

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

Proposition 7. γ and α form a Galois connection and there exists Galois connections:

$$\mathbf{M} \xleftrightarrow[\alpha_1]{\gamma_1} \widehat{\mathbf{M}} \xleftrightarrow[\alpha_2]{\gamma_2} \widehat{\mathbf{M}}^{fi}$$

The first Galois connection $\mathbf{M} \xleftrightarrow[\alpha_1]{\gamma_1} \widehat{\mathbf{M}}$ is justified piecewise by the Galois connections between $\mathbf{Val} \xleftrightarrow[\alpha]{\gamma} \widehat{\mathbf{Val}}$ and $\mathbf{Time} \xleftrightarrow[\alpha]{\gamma} \widehat{\mathbf{Time}}$. The second Galois connection $\widehat{\mathbf{M}} \xleftrightarrow[\alpha_2]{\gamma_2} \widehat{\mathbf{M}}^{fi}$ is justified by calculation over their defi-

nitions. We aim to recover this proof more easily through compositional components in Section 8.

Corollary 1.

$$\Sigma \xleftrightarrow[\alpha_1]{\gamma_1} \widehat{\Sigma} \xleftrightarrow[\alpha_2]{\gamma_2} \widehat{\Sigma}^{fi}$$

This property is derived by transporting each Galois connection between monads through their respective Galois connections to Σ . This is also a direct corollary of Theorem 1 in Section 8.5 when cast in the compositional framework of Galois transformers.

Proposition 8. *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 an application of the monotonicity of *step* and the Galois connections between monads, each transported through the Galois connection to their corresponding transition systems.

We note that the implementation for our interpreter and abstract garbage collector remain the same for each instantiation; they scale seamlessly to variations in path and flow sensitivity 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 relating $\widehat{\mathbf{M}}$ to \mathbf{M} . 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 require. Toward this goal we introduce a compositional framework for constructing monads which are correct-by-construction by extending the well-known structure of monad transformer to that of *Galois transformer*.

There are two monadic effects used in our monadic interpreter: state and nondeterminism. For state we will review the state monad transformer $S_t[s]$, which is standard (See Liang et al. [11] for more details). For nondeterminism we develop two monad transformers, \mathcal{P}_t and $FS_t[s]$. These transformers are fully general purpose, even outside the context of program analysis, and are novel in this work.

To create a monad with various state and nondeterminism effects, one must merely summon some composition of these three monads. *Implementations and proofs for monadic sequencing, state effects, nondeterminism effects, and mappings to an executable transition system will come entirely for free.* This means that for a language which has a different state space than the example in this paper, no added effort is required to construct a monad stack for that language.

Path and flow sensitivity properties arise from the *order of composition* of monad transformers. Placing state after nondeterminism ($S_t[s] \circ \mathcal{P}_t$ or $S_t[s] \circ FS_t[s']$) will result in s being path-sensitive. Placing state before nondeterminism ($\mathcal{P}_t \circ S_t[s]$ or $FS_t[s'] \circ S_t[s]$) will result in s being flow-

insensitive. Finally, when $FS_t[s]$ is used in place of $S_t[s] \circ \mathcal{P}_t$ or $\mathcal{P}_t \circ S_t[s]$, s will be flow-sensitive. The combination of all three sensitivities looks like $(M := S_t[s_1] \circ FS_t[s_2] \circ S_t[s_3])$, which will induce state space transition system $\Sigma(Exp) := [(Exp \times s_1) \mapsto s_2] \times s_3$, which is path-sensitive in s_1 , flow-sensitive in s_2 and flow-insensitive in s_3 . Using $S_t[s]$, \mathcal{P}_t and $FS_t[s]$, one can easily choose which components of the analysis are path-sensitive, flow-sensitive or flow-insensitive.

In the following definitions we must necessarily use *bind*, *return* and other operations from the underlying monad, and we notate these $bind_m$, $return_m$, \mathbf{do}_m , \leftarrow_m , etc.

8.1 State Monad Transformer

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

$$\begin{aligned} S_t[_] &: (Type \rightarrow Type) \rightarrow (Type \rightarrow Type) \\ S_t[s](m)(\alpha) &:= s \rightarrow m(\alpha \times s) \end{aligned}$$

$S_t[s]$ transports monad operations from m to $S_t[s](m)$:

$$\begin{aligned} bind &: \forall \alpha \beta, \\ &S_t[s](m)(\alpha) \rightarrow (\alpha \rightarrow S_t[s](m)(\beta)) \rightarrow S_t[s](m)(\beta) \\ bind(m)(f)(s) &:= (x, s') \leftarrow_m m(s) ; f(x)(s') \\ return &: \forall \alpha, \alpha \rightarrow S_t[s](m)(\alpha) \\ return(x)(s) &:= return_m(x, s) \end{aligned}$$

$S_t[s]$ supports state effects:

$$\begin{aligned} get &: S_t[s](m)(s) & get(s) &:= return_m(s, s) \\ put &: s \rightarrow S_t[s](m)(1) & put(s')(s) &:= return_m(1, s') \end{aligned}$$

Finally, $S_t[s]$ transports nondeterminism effects from m to $S_t[s](m)$:

$$\begin{aligned} mzero &: \forall \alpha, S_t[s](m)(\alpha) \\ mzero(s) &:= mzero_m \\ _ \langle + \rangle _ &: \forall \alpha, S_t[s](m)(\alpha) \times S_t[s](m)(\alpha) \rightarrow S_t[s](m)(\alpha) \\ (m_1 \langle + \rangle m_2)(s) &:= m_1(s) \langle + \rangle_m m_2(s) \end{aligned}$$

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 resulted in operations which do not respect monad laws or nondeterminism effect laws—ours respects both.

The nondeterminism monad transformer is defined with the expected type, embedding \mathcal{P} inside m :

$$\begin{aligned} \mathcal{P}_t &: (Type \rightarrow Type) \rightarrow (Type \rightarrow Type) \\ \mathcal{P}_t(m)(\alpha) &:= m(\mathcal{P}(\alpha)) \end{aligned}$$

\mathcal{P}_t transports monad operations from m to $\mathcal{P}_t(m)$ provided that m is a join-semilattice functor. The join-lattice functoriality of m will be instantiated with $\mathcal{P}(\beta)$.

$$\begin{aligned} bind &: \forall \alpha \beta, \mathcal{P}_t(m)(\alpha) \rightarrow (\alpha \rightarrow \mathcal{P}_t(m)(\beta)) \rightarrow \mathcal{P}_t(m)(\beta) \\ bind(m)(f) &:= \\ \{x_1..x_n\} &\leftarrow_m m ; f(x_1) \sqcup_m \dots \sqcup_m f(x_n) \end{aligned}$$

$$\begin{aligned} return &: \forall \alpha, \alpha \rightarrow \mathcal{P}_t(m)(\alpha) \\ return(x) &:= return_m(\{x\}) \end{aligned}$$

\mathcal{P}_t transports state effects from m to $\mathcal{P}_t(m)$:

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

Finally, \mathcal{P}_t supports nondeterminism effects through a straightforward application of the underlying monad's join-semilattice functoriality:

$$\begin{aligned} mzero &: \forall \alpha, \mathcal{P}_t(m)(\alpha) \\ mzero &:= \perp_m \\ _ \langle + \rangle _ &: \forall \alpha, \mathcal{P}_t(m)(\alpha) \times \mathcal{P}_t(m)(\alpha) \rightarrow \mathcal{P}_t(m)(\alpha) \\ m_1 \langle + \rangle m_2 &:= m_1 \sqcup_m m_2 \end{aligned}$$

Proposition 9. (1) *bind* and *return* satisfy monad laws, (2) *get* and *put* satisfy the state monad laws, and (3) *mzero* and $\langle + \rangle$ satisfy the nondeterminism monad laws.

The key lemma in (1) is the functoriality of m , namely that:

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

(2) is by simple calculation and (3) is trivial as a consequence of the underlying monad being a join-semilattice functor.

8.3 Flow Sensitivity Monad Transformer

The flow sensitivity monad transformer is a unique monad transformer that combines state and nondeterminism effects, and does not arise naturally from composing vanilla nondeterminism and state transformers. The flow sensitivity monad transformer is defined:

$$\begin{aligned} FS_t[_] &: (Type \rightarrow Type) \rightarrow (Type \rightarrow Type) \\ FS_t[s](m)(\alpha) &:= s \rightarrow m([\alpha \mapsto s]) \end{aligned}$$

where $[\alpha \mapsto s]$ is notation for a finite map from α to s .

$FS_t[s]$ transports monad operations provided that s is a join-semilattice and m is a join-semilattice functor. The functoriality of m will be instantiated with $[\beta \rightarrow s]$, which forms a lattice when s does also.

$$\begin{aligned} bind &: \forall \alpha \beta, \\ &FS_t[s](m)(\alpha) \rightarrow (\alpha \rightarrow FS_t[s](m)(\beta)) \rightarrow FS_t[s](m)(\beta) \\ bind(m)(f)(s) &:= \mathbf{do}_m \\ &\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\} \leftarrow_m m(s) \\ &f(x_1)(s_1) \sqcup_m \dots \sqcup_m f(x_n)(s_n) \\ return &: \forall \alpha, \alpha \rightarrow FS_t[s](m)(\alpha) \\ return(x)(s) &:= return_m\{x \mapsto s\} \end{aligned}$$

$FS_t[s]$ transports state effects from m to $FS_t[s](m)$:

$$\begin{aligned} get &: FS_t[s](m)(s) \\ get(s) &:= return_m\{s \mapsto s\} \end{aligned}$$

$$\begin{aligned} \text{put} : s &\rightarrow FS_t[s](m)(1) \\ \text{put}(s')(s) &:= \text{return}_m\{1 \mapsto s'\} \end{aligned}$$

Finally, $FS_t[s]$ supports nondeterminism effects provided s is a join-semilattice and m is a join-semilattice functor:

$$\begin{aligned} \text{mzero} : \forall \alpha, FS_t[s](m)(\alpha) \\ \text{mzero}(s) &:= \perp_m \\ _ \langle + \rangle _ : \forall \alpha, FS_t[s](m)(\alpha) \times FS_t[s](m)(\alpha) &\rightarrow FS_t[s](m)(\alpha) \\ (m_1 \langle + \rangle m_2)(s) &:= m_1(s) \sqcup_m m_2(s) \end{aligned}$$

Proposition 10. *get and put satisfy the state monad laws, mzero and $\langle + \rangle$ satisfy the nondeterminism monad laws, and $S_t[s] \circ \mathcal{P}_t \xrightarrow[\alpha_1]{\gamma_1} FS_t[s] \xrightarrow[\alpha_2]{\gamma_2} \mathcal{P}_t \circ S_t[s]$.*

These proofs are analogous to those for state and nondeterminism monad transformers.

8.4 Mapping to State Spaces

Both our execution and correctness frameworks require that monadic actions in m map to state space transitions in Σ . We extend the earlier statement of Galois connection to the transformer setting, mapping monad transformer actions in T to state space functor transitions in Π .

$$\begin{aligned} T : (Type \rightarrow Type) &\rightarrow (Type \rightarrow Type) \\ \Pi : (Type \rightarrow Type) &\rightarrow (Type \rightarrow Type) \\ \text{mstep} : \forall \alpha \beta, \end{aligned}$$

$$(\alpha \rightarrow T(m)(\beta)) \xrightarrow[\alpha]{\gamma} (\Pi(\Sigma_m)(\alpha) \rightarrow \Pi(\Sigma_m)(\beta))$$

In the type of mstep , m is an arbitrary monad whose monadic actions map to state space Σ_m . The monad transformer T must induce a state space transformer Π for which mstep can be defined. 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:

$$\begin{aligned} \text{mstep-}\gamma : \forall \alpha \beta, \\ (\alpha \rightarrow S_t[s](m)(\beta)) &\rightarrow (\Sigma_m(\alpha \times s) \rightarrow \Sigma_m(\beta \times s)) \\ \text{mstep-}\gamma(f) &:= \text{mstep}_m\text{-}\gamma(\lambda(a, s).f(a)(s)) \end{aligned}$$

For the nondeterminism transformer \mathcal{P}_t mstep is defined:

$$\begin{aligned} \text{mstep-}\gamma : \forall \alpha \beta, \\ (\alpha \rightarrow \mathcal{P}_t(m)(\beta)) &\rightarrow (\Sigma_m(\mathcal{P}(\alpha)) \rightarrow \Sigma_m(\mathcal{P}(\beta))) \\ \text{mstep-}\gamma(f) &:= \text{mstep}_m\text{-}\gamma(F) \text{ where} \\ F(\{x_1..x_n\}) &= f(x_1) \sqcup_m \dots \sqcup_m f(x_n) \end{aligned}$$

For the flow sensitivity monad transformer $FS_t[s]$ mstep is defined:

$$\begin{aligned} \text{mstep-}\gamma : \forall \alpha \beta, \\ (\alpha \rightarrow FS_t[s](m)(\beta)) &\rightarrow (\Sigma_m([\alpha \mapsto s]) \rightarrow \Sigma_m([\beta \times s])) \\ \text{mstep-}\gamma(f) &:= \text{mstep}_m\text{-}\gamma(F) \text{ where} \\ F(\{x_1 \mapsto s_1\}, \dots, \{x_n \mapsto s_n\}) &:= \\ f(x_1)(s_1) \sqcup_m \dots \sqcup_m f(x_n)(s_n) \end{aligned}$$

The Galois connections for mstep for $S_t[s]$, \mathcal{P}_t and $FS_t[s]$ crucially require that $\text{mstep}_m\text{-}\gamma$ and $\text{mstep}_m\text{-}\alpha$ be homomorphic, i.e. that:

$$\alpha(id) \sqsubseteq \text{return} \text{ and } \alpha(f \circ g) \sqsubseteq \alpha(f) \langle \circ \rangle \alpha(g)$$

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

8.5 Galois Transformers

The capstone of our framework is the fact that monad transformers $S_t[s]$, $FS_t[s]$ and \mathcal{P}_t are also *Galois transformers*.

Definition 1. *A monad transformer T is a Galois transformer if:*

1. *It transport Galois connections, that is for all monads m_1 and m_2 , $m_1 \xrightarrow[\alpha]{\gamma} m_2$ implies $T(m_1) \xrightarrow[\alpha]{\gamma} T(m_2)$.*

$$\begin{array}{ccc} m_1 & \xrightarrow{T} & T(m_1) \\ \gamma \uparrow & \alpha & \uparrow \gamma_T \\ m_2 & \xrightarrow{T} & T(m_2) \end{array}$$

2. *It transports mappings to an executable transition system, that is there exists Π s.t. for all monads m and functors Σ , $(\alpha \rightarrow m(\beta)) \xrightarrow[\alpha]{\gamma} (\Sigma(\alpha) \rightarrow \Sigma(\beta))$ implies $(\alpha \rightarrow T(m)(\beta)) \xrightarrow[\alpha]{\gamma} (\Pi(\Sigma)(\alpha) \rightarrow \Pi(\Sigma)(\beta))$.*

$$\begin{array}{ccc} \alpha \rightarrow m(\beta) & \xrightarrow{T} & \alpha \rightarrow T(m)(\beta) \\ \gamma \uparrow & \alpha & \uparrow \gamma_T \\ \Sigma(\alpha) \rightarrow \Sigma(\beta) & \xrightarrow{\Pi} & \Pi(\Sigma)(\alpha) \rightarrow \Pi(\Sigma)(\beta) \end{array}$$

Property (1) transports Galois connections between monads, and property (2) transports Galois connections between transition systems. By composing the 2-dimensional diagrams (1) and (2) into a 3-dimensional diagram (which we do not show) we establish the following theorem:

Theorem 1. *If T is a Galois transformer, then it is sufficient to prove that underlying monads m_1 and m_2 form a Galois connection $m_1 \xrightarrow[\alpha]{\gamma} m_2$ in order to establish $\Pi(\Sigma_1) \xrightarrow[\alpha]{\gamma} \Pi(\Sigma_2)$.*

This is the workhorse of our entire proof framework, allowing us to reason about monadic actions, like the monadic interpreter *step* from section 5, and derive properties about the induced transition system, which is how the analysis is executed, for free.

Proposition 11. *$S_t[s]$, $FS_t[s]$ and \mathcal{P}_t are Galois transformers.*

The proofs are sketched earlier in Section 8.

8.6 Building Transformer Stacks

We can now build monad transformer stacks from combinations of $S_t[s]$, $FS[s]_t$ and \mathcal{P}_t which automatically construct 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 $\Sigma \rightarrow \Sigma$ for some Σ , allowing one to execute the analysis.
- Galois connections between Σ and $\widehat{\Sigma}$ are established piecewise from monad transformer components.

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

$$\begin{array}{c|c|c} S_t[\widehat{\text{Env}}] & S_t[\widehat{\text{Env}}] & S_t[\widehat{\text{Env}}] \\ S_t[\widehat{\text{KAddr}}] & S_t[\widehat{\text{KAddr}}] & S_t[\widehat{\text{KAddr}}] \\ S_t[\widehat{\text{KStore}}] & S_t[\widehat{\text{KStore}}] & S_t[\widehat{\text{KStore}}] \\ S_t[\widehat{\text{Time}}] & S_t[\widehat{\text{Time}}] & S_t[\widehat{\text{Time}}] \\ S_t[\widehat{\text{Store}}] & S_t[\widehat{\text{Store}}] & \mathcal{P}_t \\ \mathcal{P}_t & FS_t[\widehat{\text{Store}}] & S_t[\widehat{\text{Store}}] \end{array}$$

From left to right these give path-sensitive, flow-sensitive and flow-insensitive analyses for the data-store.

Another benefit of our approach is that we can easily select different path and flow sensitivity properties for the data-store and stack-store independent of each other, merely by rearranging the order of composition.

$$\begin{array}{c|c|c} S_t[\widehat{\text{Env}}] & S_t[\widehat{\text{Env}}] & S_t[\widehat{\text{Env}}] \\ S_t[\widehat{\text{KAddr}}] & S_t[\widehat{\text{KAddr}}] & S_t[\widehat{\text{KAddr}}] \\ S_t[\widehat{\text{Time}}] & S_t[\widehat{\text{Time}}] & S_t[\widehat{\text{Time}}] \\ S_t[\widehat{\text{KStore}}] & S_t[\widehat{\text{KStore}}] & \mathcal{P}_t \\ \mathcal{P}_t & FS_t[\widehat{\text{KStore}}] & S_t[\widehat{\text{KStore}}] \\ S_t[\widehat{\text{Store}}] & S_t[\widehat{\text{Store}}] & S_t[\widehat{\text{Store}}] \end{array}$$

From left to right these give analysis which are all flow-insensitive in the data-store, but path-sensitive, flow-sensitive and flow-insensitive in the stack-store.

9. Implementation

We have implemented our framework in Haskell and applied it to compute analyses for λIF . 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 implementation is suitable for prototyping and exploring the design space of static analyzers. Our analyzer supports exponentially more compositions of analysis features than any current analyzer. For example, our implementation is the first which can combine arbitrary choices in call-site, object, path and flow sensitivities. Furthermore, the user can choose different path and flow sensitivities for each component of the state space.

Our implementation `maam` supports command-line flags for garbage collection, mCFA, call-site sensitivity, object sensitivity, and path and flow sensitivity.

```
./maam prog.lam --gc --mcfa --kcfa=1 --ocfa=2
```

```
--data-store=flow-sen --stack-store=path-sen
```

These flags are implemented completely independently of one another and their combination is applied to a single parameterized monadic interpreter. Furthermore, using Galois transformers allows us to prove each combination correct in one fell swoop.

A developer wishing to use our library to develop analyzers for their language of choice inherits as much of the analysis infrastructure as possible. We provide call-site, object, path and flow sensitivities as language-independent libraries. To support analysis for a new language a developer need only implement:

- A monadic semantics for their language, using state and nondeterminism effects.
- The abstract value domain, and optionally the concrete value domain if they wish to recover concrete execution.
- Intentional optimizations for their semantics like garbage collection and mcfa.

The developer then receives the following for free through our analysis library:

- A family of monads which implement their effect interface and give different path and flow sensitivities.
- An execution engine for each monad to drive the analysis.
- Mechanisms for call-site and object sensitivities.

Not only is a developer able to reuse our implementation of call-site, object, path and flow sensitivities, they need not understand the execution machinery or soundness proofs for them either. They need only verify that their monadic semantics is monotonic w.r.t. the analysis parameters, and that their abstract value domain forms a Galois connection. The execution and correctness of the final analyzer is constructed automatically given these two properties.

Our implementation is publicly available and can be installed as a cabal package: `cabal install maam`.

10. Related Work

Overview Program analysis comes in many forms such as points-to [1], flow [9], or shape analysis [2], and the literature is vast. (See Hind [8], Midtgaard [12] 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 Milanova et al. [15], Nielson and Nielson [17], Shivers [20], Van Horn and Might [22]; there are many more). These parameters come in various forms with overloaded meanings such as object [15, 21], context [19, 20], path [6], and heap [22] sensitivities, or some combination thereof [10].

These various forms can all be cast in the theory of abstraction interpretation of Cousot and Cousot [4, 5] 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 is inspired by the trifecta combination of Cousot and Cousot’s theory of abstract interpretation based on Galois connections [3–5], Moggi’s original monad transformers [16] which were later popularized in Liang et al.’s *Monad Transformers and Modular Interpreters* [11], and Sergey et al.’s *Monadic Abstract Interpreters* [18].

Liang et al. [11] 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 also extends to *abstract* interpreter construction using Galois transformers. 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 [3] 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 from 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.

We build directly on the work of Abstracting Abstract Machines (AAM) by Van Horn and Might [22] and Smaragdakis et al. [21] in our parameterization of abstract time to achieve call-site and object 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 added cost of proving that the instrumented semantics simulate the original concrete semantics.

Monadic Abstract Interpreters Sergey et al. first introduced the concept of writing abstract interpreters in monadic style in *Monadic Abstract Interpreters* (MAI)[18], in which variations in analysis are also recovered through new monad implementations. However, our approach is considerably different from MAI.

In MAI, the framework’s interface is based on *denotation functions* for every syntactic form of the language

(See “CPSInterface”, Figure 2 in MAI). This design decision has far reaching consequences for the entire approach. The denotation functions in MAI are language-specific and specialized to their example language. MAI uses a single monad stack fixed to the denotation function interface: state on top of list (Section 5.3.1 in MAI). New analyses are achieved through multiple denotation functions into this single monad. Analyses in MAI are all fixed to be path-sensitive, and the methodology for incorporating other path or flow properties is to surgically instrument the execution of the analysis with a custom Galois connection (Section 6.5 in MAI). Lastly, the framework provides no reasoning principles or proofs of soundness for the denotation function interface. *A user of MAI must inline the definitions of each analysis and prove their implementation correct from scratch each time.*

By contrast, our framework’s interface is based on state and nondeterminism *monadic effects* (Section 4.1). This interface comes equipped with reasoning principles, allowing one to verify the correctness of a monadic interpreter *independent of a particular monad*, which is not possible in MAI. State and nondeterminism monadic effects capture the essence of *small-step relational semantics*, and are therefore truly language independent. Our tools are reusable for any semantics described as a small-step state machine relation. Because we place the monadic interpreter behind an interface of effects rather than denotation functions, we are able to introduce language-independent monads which capture flow-sensitivity and flow-insensitivity (Sections 7 and 8), and we show how to compose these features with other analysis design choices (Sections 4 and 8). The monadic effect interface also allows us to separate the monad from the abstract domain, both of which are tightly coupled in MAI. Finally, our framework is compositional through the use of monad transformers (Section 8) which construct execution engines and proofs of soundness for free.

We do not achieve correctness and compositionality *in addition* to our transition from denotation functions to monadic effects; rather we achieve correctness and compositionality *through it*; such a transition is essential, primary and novel to our work.

Widening for Control-Flow Hardekopf et al. also introduce a unifying account of control flow properties in *Widening for Control-Flow* (WCF)[7], accounting for path, flow and call-site sensitivities. WCF achieves this through an instrumentation of the abstract machine’s state space which is allowed to track arbitrary contextual information, up to the path-history of the entire execution. WCF also develops a modular proof framework, proving the bulk of soundness proofs for each instantiation of the instrumentation at once.

Our work achieves similar goals, although isolating path and flow sensitivity is not our primary objective. While WCF is based on a language-dependent instrumentation of the semantics, we achieve variations in path and flow sensitivity

by modifying language-independent control properties of the interpreter through a monad.

Particular strengths of WCF are the wide range of choices for control-flow sensitivity which are shown to be implementable within the design, and the modular proof framework. For example, WCF is able to also account for call-site sensitivity through their design; we must account for call-site sensitivity through a different mechanism.

Particular strengths of our work is the understanding of path and flow sensitivity not through instrumentation but through control properties of the interpreter, and also a modular proof framework, although modular in a different sense from WCF. We also show how to compose different path and flow sensitivity choices for independent components of the state space, like a flow-sensitive data-store and path-sensitive stack-store, for example.

11. Conclusion

We have shown that *Galois transformers*, monad transformers that transport (1) Galois connections and (2) mappings to an executable transition system, 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 facilitate the systematic construction of program analyses and bridge the gap between various communities which often work in isolation.

References

- [1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
- [2] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90. ACM, 1990.
- [3] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
- [4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.
- [5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79. ACM, 1979.
- [6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02. ACM, 2002.
- [7] B. Hardekopf, B. Wiedermann, B. Churchill, and V. Kashyap. Widening for Control-Flow. In *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014.
- [8] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2001.
- [9] N. D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1981.
- [10] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13. ACM, 2013.
- [11] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. ACM, 1995.
- [12] J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.
- [13] M. Might and O. Shivers. Improving flow analyses via Γ CFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, 2006.
- [14] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k -cfa paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2010.
- [15] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 2005.
- [16] E. Moggi. An abstract view of programming languages. Technical report, Edinburgh University, 1989.
- [17] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1997.
- [18] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013.
- [19] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*, chapter 7. Prentice-Hall, Inc., 1981.
- [20] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.
- [21] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.
- [22] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10. ACM, 2010.