Galois Transformers and Modular Abstract Interpreters

Reusable Metatheory for Program Analysis

Abstract

The design and implementation of static analyzers have becoming 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 transports 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 it 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 a 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 approach to structuring and implementing program analysis. Inspired by Liang, Hudak, and Jones's *Monad transformers for 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 Galois connections.

Most significantly, Galois transformers can be proved sound once and used everywhere. 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.

Using Galois transformers, we enable arbitrary composition of analysis parameters. Our implementation `maam` supports command-line flags for garbage collection, mCFA, call-site sensitivity, object sensitivity, and path and flow sensitivities.

```
./maam --gc --mcfa --kcfa=1 --ocfa=2
  --data-store=flow-sen --stack-store=path-sen
  prog.lam
```

These flags are implemented completely independently of one another and their systematic combination is applied to a

single parameterized monadic interpreter. Furthermore, using Galois transformers allows us to prove each combination correct in one fell swoop.

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 flow and path sensitivities 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 various 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, support arbitrary relational small-step state-machine semantics and correspond to the 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 flow and path sensitivity (Section 6).

Isolating Path and Flow Sensitivity We give specific monads for instantiating the interpreter from Section 5 which give rise 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 **??**). 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 **??**). 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 Hackage[1], Haskell's package manager.

Contributions We make the following contributions:

- A methodology for constructing monadic abstract interpreters based on *monadic effects*[2].
- A language-independent library for constructing monads which have various analysis properties based on *monad transformers*.
- A 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 monad transformers for nondeterminism which give rise naturally to path-sensitive, flow-sensitive and flow-insensitive analyses.
- An isolated understanding of flow and path (in)sensitivity for static analysis as a property 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 use the operator @ 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 and abstract instantiations for the interpretation parameters in Section 6. The variations in 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 re-

[1] http://...[redacted]...

[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.

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

Figure 1: **λIF** Syntax and Concrete State Space

cover both concrete and abstract garbage collectors through a single monadic garbage collector.

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

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

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 values.

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

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 $\kappa\sigma$:

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

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

$$
\begin{array}{l}
A[\![\_]\!] \in \mathit{Atom} \to (\mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Val}) \\
A[\![i]\!](\rho, \sigma) := i \\
A[\![x]\!](\rho, \sigma) := \sigma(\rho(x)) \\
A[\![\underline{\boldsymbol{\lambda}}(x).e]\!](\rho, \sigma) := \langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle \\
 \\
\delta[\![\_]\!] \in \mathit{IOp} \to (\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 \\
 \\
\_ \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 \\
\qquad \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', \tau + 1 \rangle \\
\qquad \text{where} \\
\qquad\quad (\langle \square \odot e \rangle, \kappa l') := \kappa\sigma(\kappa l) \\
\qquad\quad \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 \leadsto \langle e, \rho'', \sigma', \kappa l', \kappa\sigma, \tau + 1 \rangle \\
\qquad \text{where} \\
\qquad\quad (\langle \langle \underline{\boldsymbol{\lambda}}(x).e, \rho' \rangle \; @ \; \square \rangle, \kappa l') := \kappa\sigma(\kappa l) \\
\qquad\quad \rho'' := \rho'[x \mapsto (x, \tau)] \\
\qquad\quad \sigma' := \sigma[(x, \tau) \mapsto A[\![a]\!](\rho, \sigma)] \\
\langle i_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto \langle i, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1 \rangle \\
\qquad \text{where} \\
\qquad\quad (\langle i_1 \oplus \square \rangle, \kappa l') := \kappa\sigma(\kappa l) \\
\qquad\quad 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 \\
\qquad \text{where} \\
\qquad\quad (\langle \underline{\textbf{if0}}(\square)\{e_1\}\{e_2\} \rangle, \kappa l') := \kappa\sigma(\kappa l) \\
\qquad\quad e := e_1 \;\; \text{when} \;\; i = 0 \\
\qquad\quad e := e_2 \;\; \text{when} \;\; i \neq 0
\end{array}
$$

Figure 2: Concrete Semantics

$$
\begin{array}{l}
\_ \leadsto^{gc} \_ \in \mathcal{P}(\Sigma \times \Sigma) \\
\varsigma \leadsto^{gc} \varsigma' \\
\qquad \text{where} \;\; \varsigma \leadsto \varsigma' \\
\langle e, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \leadsto^{gc} \langle e, \rho, \sigma', \kappa l, \kappa\sigma', \tau \rangle \\
\qquad \text{where} \\
\qquad\quad \sigma' := \{l \mapsto \sigma(l) \mid l \in R(\sigma, \rho, e)\} \\
\qquad\quad \kappa\sigma' := \{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in \mathit{KR}(\kappa\sigma, \kappa l)\}
\end{array}
$$

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

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

where ς_0 is the injection of the initial program e_0:

$$\varsigma_0 \coloneqq \langle e_0, \bot, \bot, 0, \bot, 1 \rangle$$

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

The term "flow" is heavily overloaded in static analysis. In this paper we identify three types of analysis flow:

1. Path sensitivity
2. Flow sensitivity
3. Flow insensitivity

Our framework exposes the essence of analysis flow (Sections 7 and 8), and therefore allows for many other choices in addition to these three. However, these properties occur frequently in the literature and have well-understood definitions, so we restrict our discussion to them.

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 will track both data and control flow precisely. At program points 3 and 4 the analysis considers separate worlds:

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

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

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

At program point 7 the analysis correctly corrolates the values of x and y:

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

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

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

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:

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

The analysis must then explore both branches at program point 6 resulting in no corrolation between values for x and y:

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

Flow-Insensitive A flow-insensitive analysis will collect 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 each flow property as a purely orthogonal parameter to the abstract interpreter. Flow properties will 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 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 flow property 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 will be designed such that variations in these parameters will recover both concrete and a family of abstract interpreters. To do this we extend the ideas developed in Van Horn and Might [22] with a new parameter for path and flow sensitivity. When finished, we will recover both the concrete semantics and a family of abstractions through instantiations of these parameters.

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

1. The monad, novel in this work, is the execution engine of the interpreter and captures 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 for the generic monadic interpreter. We 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 its 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: Env, $Store$, $KAddr$ and $KStore$. The nondeterminism effect will mediate branching in the execution of the interpreter. Our result is that path and flow sensitivities can be recovered by altering how these effects interact in the monad.

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

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

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

We use monad laws (left and right units, and associativity) to reason about our interpreter in the absence of a particular implementation of $bind$ and $return$. As is traditional with monadic programming, we use semicolon notation as syntactic sugar for $bind$. For example: $a \leftarrow m \ ; \ k(a)$ is just sugar for $bind(m)(k)$. We replace semicolons with line breaks headed by a $\mathbf{do}$ 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.

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

We use the state monad laws to reason about state effects, and we refer the reader to Liang et al. [11] for the definitions.

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

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

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 $State(\alpha)$ is defined as $s \to (\alpha \times s)$ and supports monadic sequencing ($bind$ and $return$) and state effects (get and put). The nondeterminism monad $Nondet(\alpha)$ is defined as $\mathcal{P}(\alpha)$ and supports monadic sequencing ($bind$ and $return$) and nondeterminism effects ($\_\langle + \rangle\_$ and $mzero$).

The combined interface of monadic sequencing, state and nondeterminism captures the abstract essence of definitions which use explicit state-passing and set comprehensions. Our interpreter will be defined up to this effect interface and avoid referencing an explicit configuration ς or explicit collections of results. This level of indirection will they be exploited: different monads will meet the same effect interface, but yield different analysis properties.

4.2 The Abstract Domain

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

The abstract domain is encapsulated by 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 $\bot$ and $\sqcup$ respecting the usual laws. We require Val to be a join-semilattice so it can be merged in the $Store$ to preserve soundness.

$$\begin{aligned} \bot &: \mathit{Val} \\ \_\sqcup\_ &: \mathit{Val} \times \mathit{Val} \to \mathit{Val} \end{aligned}$$

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 comparing with zero.

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

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

$$\begin{aligned} &\{\mathbf{true}\} \sqsubseteq \mathit{int\text{-}if0\text{-}E}(\mathit{int\text{-}I}(i)) \ \mathbf{if}\ i = 0 \\ &\{\mathbf{false}\} \sqsubseteq \mathit{int\text{-}if0\text{-}E}(\mathit{int\text{-}I}(i)) \ \mathbf{if}\ i \neq 0 \\ &\bigsqcup_{b \in \mathit{int\text{-}if0\text{-}E}(v), i \in \theta(b)} \mathit{int\text{-}I}(i) \sqsubseteq v \end{aligned}$$

where

$$\begin{aligned} \theta(\mathbf{true}) &= \{0\} \\ \theta(\mathbf{false}) &= \{i \mid i \in \mathbb{Z} \ ; \ i \neq 0\} \end{aligned}$$

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

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

Finally, δ must be sound and complete w.r.t. the abstract semantics:

$$int\text{-}I(i_1 + i_2) \sqsubseteq \delta[\![+]\!](int\text{-}I(i_1), int\text{-}I(i_2))$$
$$int\text{-}I(i_1 - i_2) \sqsubseteq \delta[\![-]\!](int\text{-}I(i_1), int\text{-}I(i_2))$$
$$\bigsqcup_{b_1 \in int\text{-}if0\text{-}E(v_1), b_2 \in int\text{-}if0\text{-}E(v_2), i \in \theta(b_1,b_2)} int\text{-}I(i) \sqsubseteq \delta[\![\odot]\!](v_1, v_2)$$

where

$$\theta(\mathbf{true}, true) = \{0\}$$
$$\theta(\mathbf{true}, \mathbf{false}) = \{i \mid i \in \mathbb{Z}\ ;\ i \neq 0\}$$
$$\theta(\mathbf{false}, \mathbf{true}) = \{i \mid i \in \mathbb{Z}\ ;\ i \neq 0\}$$
$$\theta(\mathbf{false}, false) = \mathbb{Z}$$

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, elimination and δ operators must 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 from variations in call-site sensitivity, and Smaragdakis et al. [21] which instantiates the parameter to achieve both call-site and object sensitivity.

$$Time\colon\ Type$$
$$tick\colon\ Exp \times KAddr \times Time \to Time$$

Remarkably, we need not state laws for $tick$. Our interpreter will always merge values which reside at the same address to achieve soundness. Therefore, any supplied implementations of $tick$ is valid from a soundness perspective. Different choices in $tick$ merely yield different tradoffs in precision and performance of the abstract semantics.

5. The Interpreter

We now present a generic monadic interpreter for **λIF** parameterized over M, Val and $Time$. First we implement $A[\![\_]\!]$, a *monadic* denotation for atomic expressions.

$$A[\![\_]\!] \in Atom \to M(Val)$$
$$A[\![i]\!] \coloneqq return(int\text{-}I(i))$$
$$A[\![x]\!] \coloneqq \mathbf{do}$$
$$\quad \rho \leftarrow get\text{-}Env$$
$$\quad \sigma \leftarrow get\text{-}Store$$
$$\quad \mathbf{if}\ x \in \rho$$
$$\quad\quad \mathbf{then}\ \ return(\sigma(\rho(x)))$$
$$\quad\quad \mathbf{else}\ \ return(\bot)$$
$$A[\![\underline{\boldsymbol{\lambda}}(x).e]\!] \coloneqq \mathbf{do}$$
$$\quad \rho \leftarrow get\text{-}Env$$
$$\quad return(clo\text{-}I(\langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle))$$

$get\text{-}Env$ and $get\text{-}Store$ are primitive operations for monadic state. $clo\text{-}I$ comes from the abstract domain interface.

Next we implement $step$, a *monadic* small-step function for compound expressions, shown in Figure 3. $step$ 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 also implement abstract garbage collection in a general away using the monadic effect interface:

$$gc\colon\ Exp \to M(1)$$
$$gc(e) \coloneqq \mathbf{do}$$
$$\quad \rho \leftarrow get\text{-}Env$$
$$\quad \sigma \leftarrow get\text{-}Store$$
$$\quad \kappa\sigma \leftarrow get\text{-}KStore$$
$$\quad put\text{-}Store(\{l \mapsto \sigma(l) \mid l \in R(\sigma, \rho, e))$$
$$\quad put\text{-}KStore(\{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in KR(\kappa\sigma, \kappa l)\})$$

where R and KR are as defined in Section 2.

In generalizing the semantics to account for nondeterminism, updates to both the value-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 the $\sqcup$ operator for our stores (in observation of soundness), we modify our definitions of $Store$ and $KStore$.

$$\sigma \in Store\colon\ Addr \to Val$$
$$\kappa\sigma \in KStore\colon\ KAddr \to \mathcal{P}(Frame \times KAddr)$$

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

```
step : Exp → M(Exp)
step(e1 ⊙ e2) := do
    tickM(e1 ⊙ e2)
    push(⟨□ ⊙ e2⟩)
    return(e1)
step(a) := do
    tickM(a)
    fr ← pop
    v ← A⟦a⟧
    case fr of
        ⟨□ ⊙ e⟩ → do
            push(⟨v ⊙ □⟩)
            return(e)
        ⟨v′ @ □⟩ → do
            ⟨λ(x).e, ρ′⟩ ← ↑p(clo-E(v′))
            τ ← get-Time
            σ ← get-Store
            put-Env(ρ′[x ↦ (x, τ)])
            put-Store(σ ⊔ [(x, τ) ↦ {v}])
            return(e)
        ⟨v′ ⊕ □⟩ → do
            return(δ⟦⊕⟧(v′, v))
        ⟨if0(□){e1}{e2}⟩ → do
            b ← ↑p(int-if0-E(v))
            if(b) then return(e1) else return(e2)
```

Figure 3: Monadic step function and garbage collection

```
↑p : ∀α, P(α) → M(α)
↑p({a1..an}) := return(a1) ⟨+⟩ .. ⟨+⟩ return(an)

push : Frame → M(1)
push(fr) := do
    κl ← get-KAddr
    κσ ← get-KStore
    κl′ ← get-Time
    put-KStore(κσ ⊔ [κl′ ↦ {fr :: κl}])
    put-KAddr(κl′)

pop : M(Frame)
pop := do
    κl ← get-KAddr
    κσ ← get-KStore
    fr :: κl′ ← ↑p(κσ(κl))
    put-KAddr(κl′)
    return(fr)

tickM : Exp → M(1)
tickM(e) = do
    τ ← get-Time
    κl ← get-KAddr
    put-Time(tick(e, κl, τ))
```

Figure 4: Monadic step function and garbage collection

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 $\Sigma \to \Sigma$.

For the monadic interpreter we require that monadic actions $Exp \to M(Exp)$ form a Galois connection with a transition system $\Sigma \to \Sigma$. This Galois connection serves two purposes. First, it allows us to implement the analysis by converting our interpreter to the transition system $\Sigma \to \Sigma$ 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 $(\Sigma \to \Sigma) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Exp \to M(Exp))$.

$$\mu(X).X \sqcup \varsigma_0 \sqcup \gamma(step)(X)$$

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

6. Recovering Analyses

To recover concrete and abstract interpreters we need only instantiate our generic monadic interpreter with concrete and abstract components. The concrete interpreter will recover the concrete semantics from Section 2, and through that correspondance, the soundness proof for the abstract semantics will be recovered largely for free.

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:

```
int-I : ℤ → Val
int-I(i) := {i}
int-if0-E : Val → P(Bool)
int-if0-E(v) := {true | 0 ∈ v} ⊔ {false | i ∈ v ∧ i ≠ 0}
```

and a straightforward concrete δ:

$$\delta[\![\_]\!](\_,\_)\colon \mathit{IOp} \to \mathbf{Val} \times \mathbf{Val} \to \mathbf{Val}$$
$$\delta[\![+]\!](v_1, v_2) \coloneqq \{i_1 + i_2 \mid i_1 \in v_1 \ ; \ i_2 \in v_2\}$$
$$\delta[\![-]\!](v_1, v_2) \coloneqq \{i_1 - i_2 \mid i_1 \in v_1 \ ; \ i_2 \in v_2\}$$

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

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

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

and tick is just a cons operator:

$$\mathit{tick}\colon \mathit{Exp} \times \mathbf{KAddr} \times \mathbf{Time} \to \mathbf{Time}$$
$$\mathit{tick}(e, \kappa l, \tau) \coloneqq (e, \kappa l) :: \tau$$

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

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

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

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

State effects merely return singleton sets:

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

Nondeterminism effects are implemented with set union:

$$\mathit{mzero}\colon \forall \alpha, \mathbf{M}(\alpha)$$
$$\mathit{mzero}(\psi) \coloneqq \{\}$$
$$\_\langle + \rangle\_\colon \forall \alpha, \mathbf{M}(\alpha) \times \mathbf{M}(\alpha) \to \mathbf{M}(\alpha)$$
$$(m_1 \ \langle + \rangle \ m_2)(\psi) \coloneqq m_1(\psi) \cup m_2(\psi)$$

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

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

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

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

$$\gamma\colon (\mathit{Exp} \to \mathbf{M}(\mathit{Exp})) \to (\mathbf{\Sigma} \to \mathbf{\Sigma})$$
$$\gamma(f)(e\psi^*) \coloneqq \{(e, \psi') \mid (e, \psi') \in f(e)(\psi) \ ; \ (e, \psi) \in e\psi^*\}$$
$$\alpha\colon (\mathbf{\Sigma} \to \mathbf{\Sigma}) \to (\mathit{Exp} \to \mathbf{M}(\mathit{Exp}))$$
$$\alpha(f)(e)(\psi) \coloneqq f(\{(e, \psi)\})$$

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, and therefore 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 for $\widehat{\mathbf{Val}}$ are defined:

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

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

The abstract δ operator is defined:

$$\delta\colon \mathit{IOp} \to \widehat{\mathbf{Val}} \times \widehat{\mathbf{Val}} \to \widehat{\mathbf{Val}}$$
$$\delta[\![+]\!](v_1, v_2) \coloneqq$$
$$\{i \mid 0 \in v_1 \wedge i \in v_2\}$$
$$\cup \{i \mid i \in v_1 \wedge 0 \in v_2\}$$
$$\cup \{+ \mid + \in v_1 \wedge + \in v_2\}$$
$$\cup \{- \mid - \in v_1 \wedge - \in v_2\}$$
$$\cup \{-, 0, + \mid + \in v_1 \wedge - \in v_2\}$$
$$\cup \{-, 0, + \mid - \in v_1 \wedge + \in v_2\}$$

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. $\mathbf{Val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Val}}$ *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:

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

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

$$tick\colon\ Exp \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{Time}} \to \widehat{\mathbf{Time}}$$
$$tick(e, \kappa l, \tau) = \lfloor (e, \kappa l) :: \tau \rfloor_k$$

Proposition 6. $\mathbf{Time} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Time}}$ *and* $tick$ *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.

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

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 insensitive analysis through a new definition for $M\colon \widehat{\mathbf{M}}^{fi}$. To do this we pull $\widehat{\mathbf{Store}}$ out of the powerset, exploiting its join-semilattice structure:

$$\Psi \coloneqq \widehat{\mathbf{Env}} \times \widehat{\mathbf{KAddr}} \times \widehat{\mathbf{KStore}} \times \widehat{\mathbf{Time}}$$
$$\widehat{\mathbf{M}}^{fi}(\alpha) \coloneqq \Psi \times \widehat{\mathbf{Store}} \to \mathcal{P}(\alpha \times \Psi) \times \widehat{\mathbf{Store}}$$

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

$$bind\colon \forall \alpha\beta, \widehat{\mathbf{M}}^{fi}(\alpha) \to (\alpha \to \widehat{\mathbf{M}}^{fi}(\beta)) \to \widehat{\mathbf{M}}^{fi}(\beta)$$
$$bind(m)(f)(\psi, \sigma) \coloneqq (\{bs_{11}..bs_{1m_1}..bs_{n1}..bs_{nm_n}\}, \sigma_1 \sqcup .. \sqcup \sigma_n)$$
where
$$(\{(a_1, \psi_1)..(a_n, \psi_n)\}, \sigma') \coloneqq m(\psi, \sigma)$$
$$(\{b\psi_{i1}..b\psi_{im_i}\}, \sigma_i) \coloneqq f(a_i)(\psi_i, \sigma')$$

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

$$return\colon \forall \alpha, \alpha \to \widehat{\mathbf{M}}^{fi}(\alpha)$$
$$return(a)(\psi, \sigma) \coloneqq (\{a, \psi\}, \sigma)$$

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

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

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:

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

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:

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

Proposition 7. γ *and* α *form an isomorphism, and therefore 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 piecewise 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 1.

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

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 instantiation. They scale seamlessly to path-sensitive and flow-insensitive variants when instantiated with the appropriate monad.

Recovering flow sensitivity is done through another analysis monad, which we develop in Section 8 in a more general setting.

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 will necessarily require. Toward this goal we introduce a compositional framework for constructing monads which are correct-by-construction–we extend the well-known structure of monad transformer to 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. Transformers can be composed in either direction, and the two possible directions of composition give rise naturally to path-sensitive and flow-insenstive analyses. Furthermore, our definition of nondeterminism monad transformer 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 $bind_m$, $return_m$, $\mathbf{do}_m$, $\leftarrow_m$, etc. for clarity.

8.1 State Monad Transformer

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

$$S_t[\_] \colon (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})$$
$$S_t[s](m)(\alpha) \coloneqq s \to m(\alpha \times s)$$

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

$$\begin{aligned}
&bind \colon \forall \alpha\beta, S_t[s](m)(\alpha) \to (\alpha \to S_t[s](m)(\beta)) \to S_t[s](m)(\beta)\\
&bind(m)(f)(s) \coloneqq \mathbf{do}_m\\
&\quad (x, s') \leftarrow_m m(s)\\
&\quad f(x)(s')\\
&return \colon \forall \alpha, \alpha \to S_t[s](m)(\alpha)\\
&return(x)(s) \coloneqq return_m(x, s)
\end{aligned}$$

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

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

Finally, the state monad transformer exposes get and put operations provided 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 either monad laws or nondeterminism effect laws.

Our nondeterminism monad transformer is defined with 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*:

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

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 $\mathcal{P}_t$:

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

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.

Path sensitivity arises naturally when a state transformer sits on top of a nondeterminism transformer. Flow insensitivity arises naturally when nondeterminism sits on top of state.

8.3 Mapping to State Spaces

Both our execution and correctness frameworks requires 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.

$$T\colon (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})$$
$$\Pi\colon (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})$$
$$\mathit{mstep}\colon \forall \alpha\beta m, (\alpha \to T(m)(\beta)) \xleftarrow[\alpha]{\gamma} (\Pi(\Sigma_m)(\alpha) \to \Pi(\Sigma_m)(\beta))$$

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:

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

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

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

The Galois connections for mstep for both $S_t[s]$ or P_t rely crucially on $\mathit{mstep}_m\gamma$ and $\mathit{mstep}_m\alpha$ being homomorphic, i.e. that:

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

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

Proposition 13. $S_t[s] \circ \mathcal{P}_t \xleftarrow[\alpha]{\gamma} \mathcal{P}_t \circ S_t[s]$.

The proof is by calculation after unfolding the definitions.

8.4 Flow Sensitivity Transformer

The flow sensitivity 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 transformer is defined:

$$\begin{aligned}&FS_t[\_]\colon (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})\\ &FS_t[s](m)(\alpha) \coloneqq s \to m([\alpha \mapsto s])\end{aligned}$$

where $[\alpha \mapsto s]$ is notation for a finite map over a defined domain in α.

$FS_t[s]$ is a monad when s is a join-semilattice and m is a join-semilattice functor:

$$\begin{aligned}&\mathit{bind}\colon \forall \alpha\beta,\\ &\quad FS_t[s](m)(\alpha) \to (\alpha \to FS_t[s](m)(\beta)) \to FS_t[s](m)(\beta)\\ &\mathit{bind}(m)(f)(s) \coloneqq \mathbf{do}_m\\ &\quad \{x_1 \mapsto s_1, .., x_n \mapsto s_n\} \leftarrow_m m(s)\\ &\quad f(x_1)(s_1)\ \langle+\rangle\ ..\ \langle+\rangle\ f(x_n)(s_n)\\ &\mathit{return}\colon \forall \alpha, \alpha \to FS_t[s](m)(\alpha)\\ &\mathit{return}(x)(s) \coloneqq \mathit{return}_m\{x \mapsto s\}\end{aligned}$$

$FS_t[s]$ has monadic state effects:

$$\begin{aligned}&\mathit{get}\colon FS_t[s](m)(s)\\ &\mathit{get}(s) \coloneqq \mathit{return}_m\{s \mapsto s\}\\ &\mathit{put}\colon s \to FS_t[s](m)(1)\\ &\mathit{put}(s')(s) \coloneqq \mathit{return}_m\{1 \mapsto s'\}\end{aligned}$$

$FS_t[s]$ has nondeterminism effects when s is a join-semilattice and m is a join-semilattice functor:

$$\begin{aligned}&\mathit{mzero}\colon \forall \alpha, FS_t[s](m)(\alpha)\\ &\mathit{mzero}(s) \coloneqq \bot_m\\ &\_\langle+\rangle\_\colon \forall \alpha, FS_t[s](m)(\alpha) x FS_t[s](m)(\alpha) \to FS_t[s](m)(\alpha)\\ &(m_1\ \langle+\rangle\ m_2)(s) \coloneqq m_1(s) \sqcup_m m_2(s)\end{aligned}$$

The last property required for $FS_t[s]$ to fit into our framework is to map monadic actions in $FS_t[s]$ to transitions in some state space transformer Π.

$$\begin{aligned}&\mathit{mstep}\text{-}\gamma\colon \forall \alpha\beta,\\ &\quad (\alpha \to FS_t[s](m)(\beta)) \to (\Sigma_m([\alpha \mapsto s]) \to \Sigma_m([\beta \times s]))\\ &\mathit{mstep}\text{-}\gamma(f) \coloneqq \mathit{mstep}_m\gamma(F)\\ &\quad \text{where}\ \ F(\{x_1 \mapsto s_1\}, .., \{x_n \mapsto s_n\}) \coloneqq\\ &\quad\quad f(x_1)(s_1)\ \langle+\rangle\ ..\ \langle+\rangle\ f(x_n)(s_n)\end{aligned}$$

Proposition 14. *get and put satisfy the state monad laws.*

Proposition 15. *mzero and $\langle+\rangle$ satisfy the nondeterminism monad laws.*

Proposition 16. $S_t[s] \circ \mathcal{P}_t \xleftarrow[\alpha_1]{\gamma_1} FS_t[s] \xleftarrow[\alpha_2]{\gamma_2} \mathcal{P}_t \circ S_t[s]$.

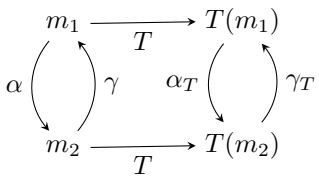
These proofs are analagous to those for state and nondeterminism monad transformers.

8.5 Galois Transformers

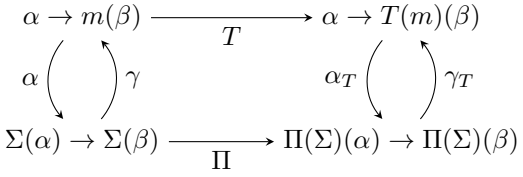
The capstone of our compositional framework is the fact that monad transformers $S_t[s]$, $FS_t[s]$ and $\mathcal{P}_t$ are also *Galois transformers*. Whereas a monad transformer is a functor between monads, a Galois transformer is a functor between Galois monads.

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

1. *For all monads m_1 and m_2, $m_1 \xleftarrow[\alpha]{\gamma} m_2$ implies $T(m_1) \xleftarrow[\alpha]{\gamma} T(m_2)$:*



2. *For all monads* m *and functors* Σ *there exists* Π *s.t.* $(\alpha \to m(\beta)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\Sigma(\alpha) \to \Sigma(\beta))$ *implies* $(\alpha \to T(m)(\beta)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\Pi(\Sigma)(\alpha) \to \Pi(\Sigma)(\beta))$*:*



Proposition 17. *$S_t[s]$, $FS_t[s]$ 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 $\Sigma \to \Sigma$ for some Σ, allowing one to execute the analysis.
- Galois connections between $\mathbf{\Sigma}$ and $\widehat{\mathbf{\Sigma}}$ are established piecewise from monad transformer components.
- Monad transformer components are proven correct for all possible languages and choices for orthogonal analysis features.

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

$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$
$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$
$S_t[\widehat{\mathbf{KStore}}]$	$S_t[\widehat{\mathbf{KStore}}]$	$S_t[\widehat{\mathbf{KStore}}]$
$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$
$S_t[\widehat{\mathbf{Store}}]$		$\mathcal{P}_t$
$\mathcal{P}_t$	$FS_t[\widehat{\mathbf{Store}}]$	$S_t[\widehat{\mathbf{Store}}]$

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:

$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$
$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$
$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$
$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$
$S_t[\mathbf{Store}]$		$\mathcal{P}_t$
$\mathcal{P}_t$	$FS_t[\mathbf{Store}]$	$S_t[\mathbf{Store}]$

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

$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$
$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$
$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$
$S_t[\mathbf{Store}]$	$FS_t[\mathbf{Store}]$	$\mathcal{P}_t$
$\mathcal{P}_t$		$S_t[\mathbf{Store}]$
$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$

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 and flow sensitivities. Furthermore, the user can choose different 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 sensitivities.

```
./maam --gc --mcfa --kcfa=1 --ocfa=2
  --data-store=flow-sen --stack-store=path-sen

  prog.lam
```

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 and flow sensitivities and 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 required effects and have different flow sensitivity properties.
- 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 and flow sensitivity, they need not understand the execution machinery or soundness proofs for them either. They need only verify that their monadic semantics is monotonic, and that their abstract value domain is sound and complete (forms a Galois connection). The execution and correctness of the final analyzer is constructed for free given these two properties.

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

```
cabal install [redacted]
```

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 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 [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 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.

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. [18] 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, 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 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 in-

terface comes equipped with reasoning principles, allowing one to verify the correctness of their 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 semanatics described as a small-step state machine. 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 completely separate the execution monad from the abstract domain, both of which are tightly coupled in the MAI approach. 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*, making such a transition essential and primary to our technique.

Unified Frameworks for Flow Sensitivity Hardekopf et al. also introduces a unifying account of flow properties in Widening for Control-Flow (WCF)[7] . 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 flow sensitivity is not our primary objective. While WCF is based on a language-dependent instrumentation of the semantics, we achieve variations in flow sensitivity by modifying control properties of the interpreter–through the monad.

Particular strengths of WCF are the wide range of choices for 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 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 make different 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 transfomers*, monad transfomers that transport 1) Galois connections and 2) mappings to transition systems, 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.

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 ΓCFA: 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*, ȦCM, 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.

A. PLDI 2015 Summary review

This paper was previously submitted to PLDI 2015. It advanced through the first round with favorable reviews, but received a short negative review in the second round stating "this paper seems like a rather straightforward and not very deep extension of the PLDI'13 paper on monadic abstract interpreters." The summary review of the PC discussion stated:

> There was extensive discussion of this paper at the PC meeting, with arguments on both sides. From the perspective of the functional programming experts, there is only a small increment from the previous PLDI'13 paper—moving from monads to monad transformers is a seemingly obvious step with little novelty. From the perspective of the static analysis experts, the paper presents a very interesting view on how to structure abstract interpreters that shows how to apply existing work in the functional world to static analysis. Ultimately we looked at the section on Galois transfomers (sic) as the point of potential novelty from the functional perspective, and found that there was too little there to convince people that Galois transformers are a non-trivial variation of monad transformers. We encourage the authors to continue this work and to try and clarify the novelty from the functional perspective (or target the paper more closely to the static analysis community). We also encourage the authors to work on the exposition in the paper to make it more accessible.

The present version of this paper has aimed to clarify the novelty of this work and explain how, while the idea of "going from monads to monad transformers" is in some sense an obvious idea, there is no way to get there from the PLDI'13 paper of Sergey et al., without significant research. That research is the subject of this paper. We have also incorporated all of the feedback toward making the paper more accessible. For completeness, we include our response to the claim "this paper seems like a rather straightforward and not very deep extension of the PLDI'13 paper on monadic abstract interpreters." Much of this content has been incorporated into the paper.

> We respectfully disagree with this assessment of the paper. Let us clarify the relationship with Sergey et al [18] (PLDI'13). The primary contributions of Sergey et al are:
>
> - Monads can serve as a good organizational mechanism for constructing abstract interpreters.
> - Different monads can be used to describe a wide range of abstract interpreters.
>
> In our view, shortcomings of Sergey et al are:
>
> - No restrictions are placed on the monads used, allowing for absurd and unsound analyses.
> - There is no soundness framework for the approach. Instantiated analyses must be reasoned about on an ad-hoc basis after inlining the abstractions.
> - The monads are not compositional or reusable across languages as claimed in the paper.
>
> Our paper remedies these shortcomings:
>
> - Our interface between interpreters and monads is well-defined and based on language-independent monadic effects, allowing one to reason on both sides of the interface.
> - Our framework constructs end-to-end Galois connections for proofs of soundness.
> - Monads in our framework can be defined and proved correct independently of a particular language. Furthermore, constructing a new large

monad in our framework is a simple composition of the monad transformers described in the paper.

In addition to these improvements, our paper has the following desirable properties:

- In our framework, intentional analysis optimizations like abstract garbage collection and shape analysis can be defined and proven correct for all concrete and abstract interpreters at once (for a particular language).
- Our framework fully compartmentalizes the choice of control sensitivity for the analysis.

Alternatively, our paper can be seen as an extension of Liang et al [11] applied to abstract interpreters rather than standard interpreters. We believe this is a deep insight into the theory of monad transformers and modular interpreters not present in Sergey et al.