Galois Transformers and Modular Abstract Interpreters

Reusable Metatheory for Program Analysis

Abstract

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

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 the 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* [1995], we propose to start with concrete interpreters 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* [Cousot and Cousot 1979] and introduce the concept of a *Galois transformer*, a monad transformer transports Galois connection.

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. For example, our implementation–called `maam` –supports command-line flags for garbage collection, k-CFA, and path- and flow-sensitivity.

```
./maam --gc --CFA=0 --flow-sen prog.lam
```

These flags are implemented completely independent of one another, and are 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). Interpreters written in this style can be reasoned about using laws that must hold for each of these interfaces. 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, call-site-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 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-sensitive machinery for any language of interest.

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 Galois connections in addition to monadic operations. Our Galois transformer framework allows us to reason about the correctness of an abstract interpreter piecewise for each transformer in a stack. These Galois transformers are also language independent, and they can be proven correct one and for all in isolation from a particular semantics.

Implementation We implement our technique in Haskell and briefly discuss how the parameters from Section 4 translate into code (Section 9). Our implementation is publicly accessible on Hackage[1], Haskell's package manager.

Contributions We make the following contributions:

- A framework for building abstract interpreters using monad transformers.
- A framework for constructing *Galois connections* using *Galois transformers*, an extension of monad transformers which also transport Galois connections.
- A new monad transformer for nondeterminism which we show is also a Galois transformer.
- An isolated understanding of flow- and path-sensitivity for static analysis as a property of the interpreter monad.

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.

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

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

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

Before designing an abstract interpreter we first specify a formal semantics for **λIF**. Our semantics makes allocation explicit and separates value and continuation stores. We will aim to recover these semantics from our generic abstract interpreter.

The state space Σ for **λIF** is a standard CESK machine augmented with a separate store for continuation values, shown in Figure 1.

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

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

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

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

The semantics of compound expressions are given relationally via the step relation $\_ \rightsquigarrow \_$ shown in Figure 2.

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

[1] http://hackage.haskell.org/package/maam

$$\_ \rightsquigarrow \_ \in \mathcal{P}(\Sigma \times \Sigma)$$

$$\langle e_1 \odot e_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \rightsquigarrow \langle e_1, \rho, \sigma, \tau, \kappa\sigma', \tau + 1\rangle$$
where $\kappa\sigma' \coloneqq \kappa\sigma[\tau \mapsto \langle \square \odot e_2\rangle :: \kappa l]$

$$\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \rightsquigarrow \langle e, \rho, \sigma, \tau, \kappa\sigma', \mathit{tick}(\tau)\rangle$$
where
$\langle \square \odot e\rangle :: \kappa l' \coloneqq \kappa\sigma(\kappa l)$
$\kappa\sigma' \coloneqq \kappa\sigma[\tau \mapsto \langle A[\![\rho, \sigma, a]\!] \odot \square\rangle :: \kappa l']$

$$\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \rightsquigarrow \langle e, \rho'', \sigma', \kappa l', \kappa\sigma, \tau + 1\rangle$$
where
$\langle \langle \underline{\boldsymbol{\lambda}}(x).e, \rho'\rangle \mathbin{@} \square\rangle :: \kappa l' \coloneqq \kappa\sigma(\kappa l)$
$\sigma' \coloneqq \sigma[(x, \tau) \mapsto A[\![\rho, \sigma, a]\!]]$
$\rho'' \coloneqq \rho'[x \mapsto (x, \tau)]$

$$\langle i_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \rightsquigarrow \langle i, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1\rangle$$
where
$\langle i_1 \oplus \square\rangle :: \kappa l' \coloneqq \kappa\sigma(\kappa l)$
$i \coloneqq \delta[\![\oplus, i_1, i_2]\!]$

$$\langle i, \rho, \sigma, \kappa l, \kappa\sigma, \tau\rangle \rightsquigarrow \langle e, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1\rangle$$
where
$\langle \underline{\textbf{if0}}(\square)\{e_1\}\{e_2\}\rangle :: \kappa l' \coloneqq \kappa\sigma(\kappa l)$
$e \coloneqq e_1 \;\text{ when }\; i = 0$
$e \coloneqq e_2 \;\text{ when }\; i \neq 0$

Figure 2: Concrete Step Relation

will show how to write a monadic garbage collector, recovering both concrete and abstract garbage collection in one fell swoop.

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

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

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

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

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

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

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

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

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

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

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

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

3. Flow Properties in Analysis

The term "flow" is heavily overloaded in static analysis. We wish to draw a sharper distinction on what is a flow property. In this paper we identify three types of analysis flow:

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

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

$$1\colon \text{let } x \coloneqq \underline{\textbf{if0}}(N)\{1\}\{-1\} \;;$$
$$2\colon \text{let } y \coloneqq \underline{\textbf{if0}}(N)\{1\}\{-1\} \;;$$
$$3\colon e$$

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

$$\{N = 0,\; x = 1\} \quad \{N \neq 0,\; x = -1\}$$

At program point 3 the analysis remains precise:

$$\{N = 0,\; x = 1,\; y = 1\} \quad \{N \neq 0,\; x = -1,\; y = -1\}$$

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

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

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

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

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

$\{N = ANY,\ x = \{-1, 1\}\}$ and
$\{N = ANY,\ x = \{-1, 1\},\ y = \{-1, 1\}\}$ respectively.

In our framework we capture both path- and flow-sensitivity as orthogonal parameters to our interpreter. Path-sensitivity will arise from the order of monad transformers used to construct the analysis. Flow-sensitivity will arise from the Galois connection used to map interpreters to state space transition systems.

4. Analysis Parameters

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

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 the path- and flow-sensitivity of the analysis.
2. The abstract domain, which for this language is merely the abstraction for integers.
3. Abstract Time, capturing call-site-sensitivity.

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

4.1 The Analysis Monad

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

$$\begin{aligned}
M &: \mathit{Type} \to \texttt{type} \\
s &: \mathit{Type} \\
\mathit{bind} &: \forall \alpha\beta, M(\alpha) \to (\alpha \to M(\beta)) \to M(\beta) \\
\mathit{return} &: \forall \alpha, \alpha \to M(\alpha) \\
\mathit{get} &: M(s) \\
\mathit{put} &: s \to M(1) \\
\mathit{mzero} &: \forall \alpha, M(\alpha) \\
_ \langle + \rangle _ &: \forall \alpha, M(\alpha) \times M(\alpha) \to M(\alpha)
\end{aligned}$$

Figure 3: Combined Monad Interface

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

Base Monad Operations A type operator M is a monad if it support bind, a sequencing operator, and its unit return. The monad interface is summarized in Figure 3.

We use the monad laws (left and right units and associativity) to reason about our implementation in the absence of a particular implementation of bind and return. For state, bind is a sequencer of state and return is the "no change in state" effect. For nondeterminism, bind implements a merging of multiple branches and return is the singleton branch.

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

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

We use the state monad laws to reason about state effects, for which refer the reader to Liang et al. [1995] for these definitions.

Nondeterminism Operations A type operator M support the nondeterminism effect if it supports an alternation operator $\langle + \rangle$ and its unit mzero. The nondeterminism interface is summarized in Figure 3.

Nondeterminism laws state that the monad must have a join-semilattice structure, and that mzero be a zero for bind.

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

$$
\begin{aligned}
\mathit{Val} &: \mathit{Type} \\
\bot &: \mathit{Val} \\
\_ \sqcup \_ &: \mathit{Val} \times \mathit{Val} \to \mathit{Val} \\
\mathit{int\text{-}I} &: \mathbb{Z} \to \mathit{Val} \\
\mathit{int\text{-}if0\text{-}E} &: \mathit{Val} \to \mathcal{P}(\mathit{Bool}) \\
\mathit{clo\text{-}I} &: \mathit{Clo} \to \mathit{Val} \\
\mathit{clo\text{-}E} &: \mathit{Val} \to \mathcal{P}(\mathit{Clo}) \\
\delta[\![\_,\_,\_]\!] &: \mathit{IOp} \times \mathit{Val} \times \mathit{Val} \to \mathit{Val} \\
\mathit{Time} &: \mathit{Type} \\
\mathit{tick} &: \mathit{Exp} \times \mathit{KAddr} \times \mathit{Time} \to \mathit{Time}
\end{aligned}
$$

Figure 4: Abstract Domain and Abstract Time Interfaces

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

Figure 5: Monadic denotation for atoms

4.2 The Abstract Domain

The abstract domain is encapsulated by the Val type in the semantics. To parameterize over it, we make Val opaque but require it support various operations. There is a constraint on Val its-self: it must be a join-semilattice with $\bot$ and $\sqcup$ respecting the usual laws. We require Val to be a join-semilattice so it can be merged in the Store. The interface for the abstract domain is shown in Figure 4.

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

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

Closures must follow similar laws:

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

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

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

4.3 Abstract Time

The interface for abstract time is familiar from Abstracting Abstract Machines [Van Horn and Might 2010](AAM)–which introduces abstract time as a single parameter from variations in call-site-sensitivity–and is shown in Figure 4.

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.

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, shown in Figure 5.

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

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

Next we implement step, a *monadic* small-step function for compound expressions, shown in Figure 6. step uses helper functions push and pop for manipulating stack frames:

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

and a monadic version of tick called tickM:

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

```
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 6: Monadic step function

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

```
gc : Exp → M(1)
gc(e) := do
    ρ ← get-Env
    σ ← get-Store
    κσ ← get-KStore
    put-Store({l ↦ σ(l) | l ∈ R[σ](ρ, e)})
    put-KStore({κl ↦ κσ(κl) | κl ∈ KR[κσ](κl)})
```

where R and KR are as defined in Section 2.

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

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

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 $\Sigma \to \Sigma$. We bridge this gap between monadic interpreters and transition functions with an extra constraint on the monad M. We require that monadic actions $\mathit{Exp} \to M(\mathit{Exp})$ 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.

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

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

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

6. Recovering Analyses

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

6.1 Recovering a Concrete Interpreter

For the concrete value space we instantiate *Val* to **Val**:

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

The concrete value space **Val** has straightforward introduction and elimination rules:

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

and the concrete δ you would expect:

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

Proposition 1. **Val** *satisfies the abstract domain laws shown in Section 4.2 Figure 4.*

Concrete time **Time** captures program contours as a product of *Exp* and **KAddr**:

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

and *tick* is just a cons operator:

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

For the concrete monad we instantiate M to a path-sensitive **M** which contains a powerset of concrete state space components.

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

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

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

State effects merely return singleton sets:

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

Nondeterminism effects are implemented with set union:

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

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

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 $\mathbf{M}$ instantiate 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:

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

The injection ς_0 for a program e_0 is:

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

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

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:

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

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

The abstract δ operator is defined:

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

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

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

Proposition 5. $\mathbf{Val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{\mathbf{Val}}$ *and their operations 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:

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

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

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

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

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

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

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

7. Varying Path- and Flow-Sensitivity

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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

Proposition 8. *There exists Galois connections:*

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

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

Corollary 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:*

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

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

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

8. A Compositional Monadic Framework

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

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

8.1 State Monad Transformer

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

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

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

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

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

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

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

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

8.2 Nondeterminism Monad Transformer

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

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

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

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

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

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

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

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

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

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

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

The proof is by simple calculation.

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

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

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

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

8.3 Mapping to State Spaces

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

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

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

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

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

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

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

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

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

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

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

Of all the γ mappings defined, the γ side of $commuteP$ is the only mapping that loses information in the α direction. Therefore, $mstep_{S_t[s]}$ and $mstep_{\mathcal{P}_t 1}$ are really isomorphism transformers, and $mstep_{\mathcal{P}_t 2}$ is the only Galois connection transformer. The Galois connections for $mstep$ for

both $S_t[s]$ or P_t rely crucially on $mstep_m\gamma$ and $mstep_m\alpha$ be homomorphic, i.e. that:

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

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

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

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

The proof is by consequence of $commuteP$.

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

The proof is by calculation after unfolding the definitions.

8.4 Galois Transformers

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

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

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

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

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

8.5 Building Transformer Stacks

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

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

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

$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$
$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$
$S_t[\widehat{\mathbf{KStore}}]$	$S_t[\widehat{\mathbf{KStore}}]$	$S_t[\widehat{\mathbf{KStore}}]$
$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$
$S_t[\widehat{\mathbf{Store}}]$	FS_t	FI_t
FS_t	$S_t[\widehat{\mathbf{Store}}]$	$S_t[\widehat{\mathbf{Store}}]$

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:

$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$	$S_t[\mathbf{Env}]$
$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$	$S_t[\mathbf{KAddr}]$
$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$	$S_t[\mathbf{KStore}]$
$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$	$S_t[\mathbf{Time}]$
$S_t[\mathbf{Store}]$	FS_t	FI_t
FS_t	$S_t[\mathbf{Store}]$	$S_t[\mathbf{Store}]$

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

$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$	$S_t[\widehat{\mathbf{Env}}]$
$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$	$S_t[\widehat{\mathbf{KAddr}}]$
$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$	$S_t[\widehat{\mathbf{Time}}]$
$S_t[\widehat{\mathbf{KStore}}]$	FS_t	FI_t
$S_t[\widehat{\mathbf{Store}}]$	$S_t[\widehat{\mathbf{KStore}}]$	$S_t[\widehat{\mathbf{KStore}}]$
FS_t	$S_t[\widehat{\mathbf{Store}}]$	$S_t[\widehat{\mathbf{Store}}]$

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

9. Implementation

We have implemented our framework in Haskell and applied it to compute analyses for **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 interpreter for **λIF** is parameterized as discussed in Section 4. We express a valid analysis with the following Haskell constraint:

$$\texttt{type}\ Analysis(\delta, \mu, m) :: Constraint =$$
$$(AAM(\mu), Delta(\delta), AnalysisMonad(\delta, \mu, m))$$

Constraints $AAM(\mu)$ and $Delta(\delta)$ are interfaces for abstract time and the abstract domain.
The constraint $AnalysisMonad(m)$ requires only that m has the required effects:

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

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

Using Galois transformers, we enable arbitrary composition of choices for various analysis components. For example, our implementation, called `maam` supports command-line flags for garbage collection, k-CFA, and path- and flow-sensitivity.

```
./maam --gc --CFA=0 --flow-sen prog.lam
```

These flags are implemented completely independent 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.

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

```
cabal install maam
```

10. Related Work

Program analysis comes in many forms such as points-to [Andersen 1994], flow [Jones 1981], or shape analysis [Chase et al. 1990], and the literature is vast. (See Hind [2001]; Midtgaard [2012] for surveys.) Much of the research has focused on developing families or frameworks of analyses that endow the abstraction with knobs, levers, and dials to tune precision and compute efficiently (some examples include Shivers [1991]; Nielson and Nielson [1997]; Milanova et al. [2005]; Van Horn and Might [2010]; there are many more). These parameters come in various forms with overloaded meanings such as object- [Milanova et al. 2005; Smaragdakis et al. 2011], context- [Sharir and Pnueli 1981; Shivers 1991], path- [Das et al. 2002], and heap- [Van Horn and Might 2010] sensitivities, or some combination thereof [Kastrinis and Smaragdakis 2013].

These various forms can all be cast in the theory of abstraction interpretation of Cousot and Cousot [1977, 1979] and understood as computable approximations of an underlying concrete interpreter. Our work demonstrates that if this underlying concrete interpreter is written in monadic style, monad transformers are a useful way to organize and compose these various kinds of program abstractions in a modular and language-independent way.

This work is inspired by the combination of Cousot and Cousot's theory of abstract interpretation based on Galois connections [1977; 1979; 1999], Liang et al.'s monad transformers for modular interpreters [1995] and Sergey et al.'s monadic abstract interpreters [2013], and continues in the tradition of applying monads to programming language semantics pioneered by Moggi [1989].

Liang et al. [1995] first demonstrated how monad transformers could be used to define building blocks for constructing (concrete) interpreters. Their interpreter monad $InterpM$ bears a strong resemblance to ours. We show this "building blocks" approach to interpreter construction extends to *abstract* interpreter construction, too, by using Galois transfomers. Moreover, we show that these monad transformers can be proved sound via a Galois connection to their concrete counterparts, ensuring the soundness of any stack built from sound blocks of Galois transformers. Soundness proofs of various forms of analysis are notoriously brittle with respect to language and analysis features. A reusable framework of Galois transformers offers a potential way forward for a modular metatheory of program analysis.

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

Sergey et al. [2013] first introduced Monadic Abstract Interpreters (MAI), in which interpreters are also written in monadic style and variations in analysis are recovered through new monad implementations. However, each monad in MAI is designed from scratch for a specific language to have specific analysis properties. The MAI work is analogous to monadic interpreter of Wadler [1992], in which the monad structure is monolithic and must be reconstructed for each new language feature. Our work extends the ideas in MAI in a way that isolates each parameter to be independent of others, similar to the approach of Liang et al. [1995]. We factor out the monad as a truly semantics independent feature. This factorization reveals an orthogonal tuning knob for path- and flow-sensitivity. Even more, we give the user building blocks for constructing monads that are correct and give the desired properties by construction. Our framework is also motivated by the needs of reasoning formally about abstract interpreters, no mention of which is made in MAI.

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

11. Conclusion

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

References

L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

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.

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

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

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.

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.

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.

N. D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1981.

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.

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.

J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.

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.

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

E. Moggi. An abstract view of programming languages. Technical report, Edinburgh University, 1989.

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.

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.

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

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

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, 2011.

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.

P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92. ACM, 1992.