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

Abstract

The design and implementation of static analyzers have becoming increasingly systematic. In fact, design and implementation have remained seemingly 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 (apparently) minor changes to the semantics and analyzers themselves. We finally reconcile the systematic construction of static analyzers with their proofs of soundness via a mechanistic Galois-connection-based metatheory for static analyzers.

1. Introduction

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

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

In this paper, we propose an alternative approach to structuring and implementing program analysis. We propose to use concrete interpreters in monadic style. As we show, classical program abstractions can be embodied as language-independent monads. Moreover, these abstractions can be written as monad transformers, thereby allowing their composition to achieve new forms of analysis. Most significantly, these monad transformers can be proved sound once and for all. Abstract interpreters, which take the form of monad transformer stacks coupled together with a monadic interpreter, inherit the soundness properties of each element in the stack. This approach enables reuse of abstractions across languages and lays the foundation for a modular metatheory of program analysis.

1.1 Contributions

Our contributions are:

- A compositional meta-theory framework for building correct-by-construction abstract interpreters. This framework is built using a restricted class of monad transformers.
- An isolated understanding of flow and path sensitivity for static analysis. We understand this spectrum as mere variations in the order of monad transformer composition in our framework.

1.2 Outline

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

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

Figure 1: λIF

2. Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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 addresses.

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

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 $\kappa\sigma$:

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

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

One key property of a static analysis is the way it tracks *flow*. The term "flow" is heavily overloaded in static analysis, for example CFA is literally the abbreviation of "control flow analysis". We wish to draw a sharper distinction on what is a flow property. First we identify three different types of flow in analysis:

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

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

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

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:

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

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

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

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:

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

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 world with environment:

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

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. For brevity, and lack of better terms, we will abbreviate these analyses as "path sensitive", "flow sensitive" and "flow insensitive". This is only ambiguous for "flow sensitive", as path sensitivity implies flow sensitivity, and flow insensitivity implies path insensitivity.

4. Analysis Parameters

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

First we describe the parameters to the interpreter. Then we conclude the section with an implementation which is generic to 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. This is the execution engine of the interpreter and captures the flow-sensitivity of the analysis.

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

Figure 2: Monad Interface

2. The abstract domain. For our language is merely an abstraction for integers.
3. The abstraction for time. Abstract time captures the call-site sensitivity of the analysis, as introduced by [CITE].

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 reasoning 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 is capturing 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.

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

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

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

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

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

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

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

Figure 3: State Monad Interface

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

Figure 4: Nondeterminism Interface

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

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

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

The effects for $\mathit{get}\text{-}\mathit{Store}$, $\mathit{get}\text{-}\mathit{KAddr}$ and $\mathit{get}\text{-}\mathit{KStore}$ are identical.

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

We use the nondeterminism laws to reason about nondeterminism effects:

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

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

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

Figure 5: Abstract Domain Interface

avoid referencing an explicit configuration ς or explicit collections of results.

4.2 The Abstract Domain

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

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:

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

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

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

$$
\begin{aligned}
\mathit{Time} &: \mathit{Type} \\
\mathit{tick} &: \mathit{Exp} \times \mathit{KAddr} \times \mathit{Time} \to \mathit{Time}
\end{aligned}
$$

Figure 6: Abstract Time Interface

4.3 Abstract Time

The interface for abstract time is familiar from the AAM literature and is shown in Figure 6. In traditional AAM, tick is defined to have access to all of Σ. This comes from the generality of the framework–to account for all possible tick functions. We only discuss instantiating Addr to support k-CFA, so we specialize the Σ parameter to $\mathit{Exp} \times \mathit{KAddr}$. Also in AAM is the opaque function $\mathit{alloc}\colon\ \mathit{Var} \times \mathit{Time} \to \mathit{Addr}$. Because we will only ever use the identity function for alloc, we omit its abstraction and instantiation in our development.

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

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

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

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

Next we implement $step$, a *monadic* small-step function for compound expressions:

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

$step$ uses helper functions $push$ and pop for manipulating stack frames:

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

and a monadic version of $tick$ called $tickM$:

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

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

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

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

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

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

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

To execute the interpreter we must introduce one more parameter. In the concrete semantics, execution takes the form of a least-fixed-point computation over the collecting semantics This in general requires a join-semilattice structure for some Σ and a transition function $\Sigma \to \Sigma$. We bridge this gap between monadic interpreters and transition functions with an extra constraint on the monad M. We require that monadic actions $Exp \to M(Exp)$ 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.

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

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

6. Recovering Analyses

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

6.1 Recovering a Concrete Interpreter

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

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

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

$$\begin{aligned}
&\mathit{int\text{-}I} \colon \mathbb{Z} \to \mathbf{Val} \\
&\mathit{int\text{-}I}(i) \coloneqq \{i\} \\
&\mathit{int\text{-}if0\text{-}E} \colon \mathbf{Val} \to \mathcal{P}(\mathit{Bool}) \\
&\mathit{int\text{-}if0\text{-}E}(v) \coloneqq \{\mathbf{true} \mid 0 \in v\} \cup \{\mathbf{false} \mid i \in v \wedge i \neq 0\}
\end{aligned}$$

and the concrete δ you would expect:

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

Proposition 1. $\mathbf{Val}$ *satisfies the abstract domain laws from section 4.2.*

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

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

and $tick$ is just a cons operator:

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

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

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

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:

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

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.*

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.*

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

6.2 Recovering an Abstract Interpreter

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

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

Introduction and elimination functions 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 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 from section 4.2.*

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:

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

The tick operator becomes cons followed by k-truncation:

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

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 to choices for $\widehat{\mathbf{Store}}$ (which maps to $\widehat{\mathbf{Val}}$) and $\widehat{\mathbf{Time}}$.

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

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 $\widehat{\mathbf{M}}$: $\widehat{\mathbf{M}}$. To do this we pull $\widehat{\mathbf{Store}}$ out of the powerset, exploiting its join-semilattice structure:

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

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

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

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

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

State effects $\mathit{get\text{-}Env}$ and $\mathit{put\text{-}Env}$ are also straightforward, returning one branch of nondeterminism:

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

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

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

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 $\widehat{\mathbf{M}}$ to a state space transition over $\widehat{\mathbf{\Sigma}}^{fi}$ must also compute set unions and store joins:

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

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

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

Proposition 8. *There exists Galois connections:*

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

and the following properties hold:

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

The first Galois connection $\mathbf{\Sigma} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \widehat{\mathbf{\Sigma}}$ 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 $\widehat{\mathbf{\Sigma}} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \widehat{\mathbf{\Sigma}}^{fi}$ is justified by first calculating the Galois connection between monads $\widehat{\mathbf{M}}$ and $\mathbf{M}$, and then transporting it through their respective Galois connections to $\widehat{\mathbf{\Sigma}}$ and $\widehat{\mathbf{\Sigma}}^{fi}$. These proofs are tedious calculations over the definitions which we do not repeat here. However, we will recover these proof in a later section through our compositional framework which greatly reduces the proof burden.

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

8. A Compositional Monadic Framework

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

There are two types of monadic effects used in our monadic interpreter: state and nondeterminism. Each of these effects have corresponding monad transformers. Monad transformers for state effects were described by Jones in [CITE]. 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, $S_t[s]$:

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

For monad transformers, $bind$ and $return$ will use monad operations from the underlying m, which we notate $bind_m$ and $return_m$. When writing in do-notation, we write $\texttt{do}_m$ and $\leftarrow_m$ for clarity.

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

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

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

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

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 can compose with state in both directions. Previous attempts to define a monad transformer for nondeterminism have resulted in monad operations which do not respect monad laws.

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

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

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

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

Proposition 9. *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 10. *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:

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

Proposition 11. *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:

$$\mathit{mstep} \colon \forall \alpha \beta, (\alpha \to M(\beta)) \xleftarrow[\alpha]{\gamma} (\Sigma(\alpha) \to \Sigma(\beta))$$

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:

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

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

$$
\begin{aligned}
&\mathit{mstep}_1 \gamma \colon \forall \alpha \beta m, \\
&\quad (\alpha \to \mathcal{P}_t(m)(\beta)) \to (\Sigma_m(\mathcal{P}(\alpha)) \to \Sigma_m(\mathcal{P}(\beta))) \\
&\mathit{mstep}_1 \gamma(f) \coloneqq \mathit{mstep}_m \gamma(F) \\
&\quad \text{where } F(\{x_1 .. x_n\}) = f(x_1) \langle + \rangle .. \langle + \rangle f(x_n))
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{mstep}_2 \gamma \colon \forall \alpha \beta m, \\
&\quad (\alpha \to \mathcal{P}_t(m)(\beta)) \to (\mathcal{P}(\Sigma_m(\alpha)) \to \mathcal{P}(\Sigma_m(\beta))) \\
&\mathit{mstep}_2 \gamma(f)(\{\varsigma_1 .. \varsigma_n\}) \coloneqq a\Sigma P_1 \cup .. \cup a\Sigma P_n \\
&\quad \text{where} \\
&\qquad \mathit{commuteP}\text{-}\gamma \colon \forall \alpha, \Sigma_m(\mathcal{P}(\alpha)) \to \mathcal{P}(\Sigma_m(\alpha)) \\
&\qquad a\Sigma P_i \coloneqq \mathit{commuteP}\text{-}\gamma(\mathit{mstep}_m \gamma(f)(\varsigma_i))
\end{aligned}
$$

The operation $\mathit{commuteP}\text{-}\gamma$ must be defined for the underlying Σ. In general, $\mathit{commuteP}$ 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.

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

Of all the γ mappings defined, the γ side of $\mathit{commuteP}$ is the only mapping that loses information in the α direction. Therefore, $\mathit{mstep}_{S_t[s]}$ and $\mathit{mstep}_{\mathcal{P}_t 1}$ are really isomorphism transformers, and $\mathit{mstep}_{\mathcal{P}_t 2}$ is the only Galois connection transformer. The Galois connections for mstep for both $S_t[s]$ or P_t rely crucially on $\mathit{mstep}_m \gamma$ and $\mathit{mstep}_m - \alpha$ to be functorial (i.e., homomorphic). For convenience, we name the pairing of $\mathcal{P}_t$ with $\mathit{mstep}_1\ FI_t$, and with mstep_2 FS_t for flow insensitive and flow sensitive respectively.

Proposition 12. $\Sigma_{FS_t} \xleftarrow[\alpha]{\gamma} \Sigma_{FI_t}$.

The proof is by consequence of $\mathit{commuteP}$.

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 Galois Transformers

The final piece 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 monads, a Galois transformer is a functor between Galois connections.

Definition 1. *A monad transformer T is a Galois transformer if for every $m_1 \xleftarrow[\alpha]{\gamma} m_2$, $T(m_1) \xleftarrow[\alpha]{\gamma} T(m_2)$.*

Proposition 14. *$S_t[s]$ and $\mathcal{P}_t$ are Galois transformers.*

The proofs are straightforward applications of the underlying $m_1 \xleftarrow[\alpha]{\gamma} m_2$.

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

8.5 Building Transformer Stacks

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

- The resulting monad has the combined effects of all pieces of the transformer stack.
- Actions in the resulting monad map to a state space transition system $\Sigma \to \Sigma$ for some Σ.

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

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

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

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

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

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

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

9. Implementation

We have implemented our framework in Haskell and applied it to compute analyses for **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:

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

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

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

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

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

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

```
cabal install maam
```

10. Related Work

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

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

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

of Galois transformers offers a potential way forward for a modular metatheory of program analysis.

[FIXME: Note how language independent characterizations of analyses could lead to more insights like: [Might et al. 2010]]

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

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

11. Conclusion

References

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

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

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

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

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

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

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

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

I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. 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. POPL '11. ACM, 2011.

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