Modular Metatheory for Abstract Interpreters

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

Writing abstract interpreters is hard. Writing proofs about abstract interpreters is extra hard. Modern practice in whole-program analysis requires multiple iterations in the design space of possible analyses. As we explore the design space of abstract interpreters, it would be nice if we didn't need to reprove all the properties we care about. What we lack is a reusable meta-theory for exploring the design space of *correct-by-construction* abstract interpreters.

We propose a compositional meta-theory framework for general purpose static analysis. Our framework gives the analysis designer building blocks for building correct-by-construction abstract interpreters. These building blocks are compositional, and they carry both computational and correctness properties of an analysis. For example, we are able to tune the flow and path sensitivities of an analysis in our framework with no extra proof burden. We do this by capturing the essential properties of flow and path sensitivities into plug-and-play components. Comparably, we show how to design an analysis to be correct for all possible instantiations to flow and path sensitivity.

To achieve compositionality, our framework leverages monad transformers as the fundamental building blocks for an abstract interpreter. Monad transformers snap together to form a single monad which drives interpreter execution. Each piece of the monad transformer stack corresponds to either an element of the semantics' state space or a nondeterminism effect. Variations in the transformer stack to give rise to different path and flow sensitivities for the analysis. Interpreters written in our framework are proven correct w.r.t. all possible monads, and therefore to each choice of path and flow sensitivity.

The monad abstraction provides the computational and proof properties for our interpreters, from the monad operators and laws respectively. Monad transformers are monad composition function; they consume and produce monads. We strengthen the monad transformer interface to require that the resulting monad have a relationship to a state machine transition space. We prove that a small set of monads transformers that meet this stronger interface can be used to write monadic abstract interpreters.

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 X gives the concrete semantics for a small functional language. Section [X][Monadic Interpreter] shows the full definition of a highly parameterized monadic interpreter. Section [X][Recovering Concrete and Abstract Interpreters] shows how to recover concrete and abstract interpreters. Section X shows how to manipulate the path and flow sensitivity of the interpreter through varyations in the monad. Section X demonstrates our compositional meta-theory framework built on monad transformers.

2. Semantics

To demonsrate our framework we design an abstract interpreter for a simple applied lambda calculus: λ-**IF**.

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

λ-**IF** extends traditional lambda calculus with integers, addition, subtration 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 λ-**IF**. 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 λ-**IF** is a standard CESK machine augmented with a separate store for continuation values:

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

Atomic expressions are given meaning through the denotation function $A[\![\_, \_, \_]\!]$:

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

Primitive operations are given meaning through the denotation function $\delta[\![\_, \_, \_]\!]$:

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

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

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

Our abstract intepreter will support abstract garbage collection [CITE], the concrete analogue of which is just standard garbage collection. Garbage collection is defined with a reachability function R which computes the transitively reachable address from (ρ, e) in σ:

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

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

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

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

$$
\begin{aligned}
FV &\in Exp \to \mathcal{P}(Var) \\
FV(x) &:= \{x\} \\
FV(i) &:= \{\} \\
FV([\lambda](x).e) &:= FV(e) - \{x\} \\
FV(e_1 \odot e_2) &:= FV(e_1) \cup FV(e_2) \\
FV(\underline{\mathbf{if0}}(e_1)\{e_2\}\{e_3\}) &:= FV(e_1) \cup FV(e_2) \cup FV(e_3)
\end{aligned}
$$

Analagously, KR is the set of transitively reachabel continuation addresses in $\kappa\sigma$:

$$
\begin{aligned}
KR[\_] &\in KStore \to KAddr \to \mathcal{P}(KAddr) \\
KR[\kappa\sigma](\kappa l) &:= \mu(k\theta).\kappa\theta_0 \cup \kappa\theta \cup \{\pi_2(\kappa\sigma(\kappa l)) \mid \kappa l \in \kappa\theta\}
\end{aligned}
$$

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

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

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

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

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

3. Flow Properties in Analysis

One key property of a static analysis is the way it tracks *flow*. The term "flow" is heavily overloaded in static analysis, 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 λ-**IF** (extended with let-bindings) where an analysis cannot determine the value of N:

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

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

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

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

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

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

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

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

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

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

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

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 AAM[CITE] 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.
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 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 thier laws.

4.1.1 Monad Properties

To be a monad, a type operator *M* must support the *bind* operation:

$$bind\colon \forall \alpha\beta, M(\alpha) \to (\alpha \to M(\beta)) \to M(\beta)$$

as well as a unit for *bind* called *return*:

$$return\colon \forall \alpha, \alpha \to M(\alpha)$$

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

$unit_1\colon bind(return(a))(k) = k(a)$

$unit_2\colon bind(m)(return) = m$

$assoc\colon bind(bind(m)(k_1))(k_2) = bind(m)(\lambda(a).bind(k_1(a))(k_2))$

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. These operators capture the essence of the combination of explicit state-passing and set comprehension in the interpreter. Our interpreter will use these operators and avoid referencing an explicit configuration ς or explicit collections of results.

As is traditional with monadic programming, we use do and semicolon notation as syntactic sugar for *bind*. For example:

$$\begin{aligned} &\texttt{do}\\ &\quad a \leftarrow m\\ &\quad k(a) \end{aligned}$$

and

$$a \leftarrow m \; ; \; k(a)$$

are both just sugar for

$$bind(m)(k)$$

4.1.2 Monad State Properties

Interacting with a state component like *Env* is achieved through *get-Env* and *put-Env* effects:

$$get\text{-}Env\colon M(Env)$$
$$put\text{-}Env\colon Env \to M(1)$$

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

$put\text{-}put\colon put(s_1) \; ; \; put(s_2) = put(s_2)$

$put\text{-}get\colon put(s) \; ; \; get = return(s)$

$get\text{-}put\colon s \leftarrow get \; ; \; put(s) = return(1)$

$get\text{-}get\colon s_1 \leftarrow get \; ; \; s_2 \leftarrow get \; ; \; k(s_1, s_2) = s \leftarrow get \; ; \; k(s, s)$

The effects for *get-Store*, *get-KAddr* and *get-KStore* are identical.

4.1.3 Monad Nondeterminism Properties

Nondeterminism is achieved through operators *mzero* and $\langle + \rangle$:

$$mzero\colon \forall \alpha, M(\alpha)$$
$$_{-}\langle + \rangle_{-}\colon \forall \alpha, M(\alpha) \times M(\alpha) \to M(\alpha)$$

We use the nondeterminism laws to reason about nondeterminism effects:

$\bot\text{-}\texttt{zero}_1\colon bind(mzero)(k) = mzero$

$\bot\text{-}\texttt{zero}_2\colon bind(m)(\lambda(a).mzero) = mzero$

$\bot\text{-}\texttt{unit}_1\colon mzero \langle + \rangle m = m$

$\bot\text{-}\texttt{unit}_2\colon m \langle + \rangle mzero = m$

$+\text{-}\texttt{assoc}\colon m_1 \langle + \rangle (m_2 \langle + \rangle m_3) = (m_1 \langle + \rangle m_2) \langle + \rangle m_3$

$+\text{-}\texttt{comm}\colon m_1 \langle + \rangle m_2 = m_2 \langle + \rangle m_1$

$+\text{-}\texttt{dist}\colon bind(m_1 \langle + \rangle m_2)(k) = bind(m_1)(k) \langle + \rangle bind(m_2)(k)$

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 integers consists of introduction and elimination rules:

$$int\text{-}I \colon \mathbb{Z} \to Val$$
$$int\text{-}if0\text{-}E \colon Val \to \mathcal{P}(Bool)$$

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

$$\{true\} \sqsubseteq int\text{-}if0\text{-}E(int\text{-}I(i)) \text{ if } i = 0$$
$$\{false\} \sqsubseteq int\text{-}if0\text{-}E(int\text{-}I(i)) \text{ if } i \neq 0$$
$$v \sqsupseteq \bigsqcup_{b \,\in\, int\text{-}if0\text{-}E(v)} \theta(b)$$
$$\text{where}$$
$$\theta(true) = int\text{-}I(0)$$
$$\theta(false) = \bigsqcup_{i \,\in\, \mathbb{Z} \,\mid\, i \neq 0} int\text{-}I(i)$$

Additionally we must abstract closures:

$$clo\text{-}I \colon Clo \to Val$$
$$clo\text{-}E \colon Val \to \mathcal{P}(Clo)$$

which follow similar laws:

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

The denotation for primitive operations δ must also be opaque:

$$\delta[\![\text{-}, \text{-}, \text{-}]\!] \colon IOp \times Val \times Val \to Val$$

We can also give soundness laws for δ using int-I and int-if0-E:

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

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

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

4.3 Abstract Time

The interface for abstract time is familiar from the AAM literature:

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

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 possibile *tick* functions. We only discuss instantiating *Addr* to support k-CFA, so we specialize the Σ parameter to $Exp \times KAddr$. Also in AAM is the opaque function $alloc \colon Var \times Time \to Addr$. 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 λ-**IF** paramaterized over *M*, *Val* and *Time*.

In moving our semantics to an analysis, we will need to reuse addresses in the state space. This induces *Store* and *KStore* to join when binding new values to in-use addresses. The state space for our interpreter will therefore use the following domain for *Store* and *KStore*:

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

We have already established a join-semilattice structure for *Val*. 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.

Before defining the interpreter we define some helper functions which interact with the underlying monad *M*.

First, values in $\mathcal{P}(\alpha)$ can be lifted to monadic values $M(\alpha)$ using *return* and *mzero*, which we name $\uparrow_p$:

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

Allocating addresses and updating time can be implemented using monadic state effects:

$$allocM : Var \rightarrow M(Addr)$$
$$allocM(x) := \texttt{do}$$
$$\tau \leftarrow get\text{-}Time$$
$$return(x, \tau)$$
$$\kappa allocM : M(KAddr)$$
$$\kappa allocM := \texttt{do}$$
$$\tau \leftarrow get\text{-}Time$$
$$return(\tau)$$
$$tickM : Exp \rightarrow M(1)$$
$$tickM(e) = \texttt{do}$$
$$\tau \leftarrow get\text{-}Time$$
$$\kappa l \leftarrow get\text{-}KAddr$$
$$put\text{-}Time(tick(e, \kappa l, \tau))$$

Finally, we introduce helper functions for manipulating stack frames:

$$push : Frame \rightarrow M(1)$$
$$push(fr) := \texttt{do}$$
$$\kappa l \leftarrow get\text{-}KAddr$$
$$\kappa \sigma \leftarrow get\text{-}KStore$$
$$\kappa l' \leftarrow \kappa allocM$$
$$put\text{-}KStore(\kappa \sigma \sqcup [\kappa l' \mapsto \{fr :: \kappa l\}])$$
$$put\text{-}KAddr(\kappa l')$$
$$pop : M(Frame)$$
$$pop := \texttt{do}$$
$$\kappa l \leftarrow get\text{-}KAddr$$
$$\kappa \sigma \leftarrow get\text{-}KStore$$
$$fr :: \kappa l' \leftarrow \uparrow_p (\kappa \sigma(\kappa l))$$
$$put\text{-}KAddr(\kappa l')$$
$$return(fr)$$

To implement our interpreter we define a denotation function for atomic expressions and a step function for compound expressions. The denotation for atomic expressions is written as a monadic computation from atomic expresssions to values.

$$A[\![\_]\!] \in Atom \rightarrow M(Val)$$
$$A[\![i]\!] := return(int\text{-}I(i))$$
$$A[\![x]\!] := \texttt{do}$$
$$\rho \leftarrow get\text{-}Env$$
$$\sigma \leftarrow get\text{-}Store$$
$$l \leftarrow \uparrow_p (\rho(x))$$
$$return(\sigma(x))$$
$$A[\.e]\!] := \texttt{do}$$
$$\rho \leftarrow get\text{-}Env$$
$$return(clo\text{-}I(\langle [\lambda](x).e, \rho \rangle))$$

The step function is written as a monadic computation from expressions to the next expression to evaluate, in small step style. The definition for operators is simple: it merely pushes a stack from and returns the first operand:

$$step : Exp \rightarrow M(Exp)$$
$$step(e_1 \odot e_2) := \texttt{do}$$
$$tickM(e_1 \odot e_2)$$
$$push(\langle \Box \odot e_2 \rangle)$$
$$return(e_1)$$

The definition for atomic expressions must pop and inspect the stack and perform the denotation of the operation:

$$step(a) := \texttt{do}$$
$$tickM(a)$$
$$fr \leftarrow pop$$
$$v \leftarrow A[\![a]\!]$$
$$\texttt{case } fr \texttt{ of}$$
$$\langle \Box \odot e \rangle \rightarrow \texttt{do}$$
$$push(\langle v \odot \Box \rangle)$$
$$return(e)$$
$$\langle v' @ \Box \rangle \rightarrow \texttt{do}$$
$$\langle [\lambda](x).e, \rho' \rangle \leftarrow \uparrow_p (clo\text{-}E(v'))$$
$$l \leftarrow alloc(x)$$
$$\sigma \leftarrow get\text{-}Store$$
$$put\text{-}Env(\rho'[x \mapsto l])$$
$$put\text{-}Store(\sigma[l \mapsto v])$$
$$return(e)$$
$$\langle v' \oplus \Box \rangle \rightarrow \texttt{do}$$
$$return(\delta(\oplus, v', v))$$
$$\langle \underline{if0}(\Box)\{e_1\}\{e_2\} \rangle \rightarrow \texttt{do}$$
$$b \leftarrow \uparrow_p (int\text{-}if0\text{-}E(v))$$
$$\texttt{if}(b) \texttt{ then } return(e_1) \texttt{ else } return(e_2)$$

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

$$gc : Exp \rightarrow M(1)$$
$$gc(e) := \texttt{do}$$
$$\rho \leftarrow get\text{-}Env$$
$$\sigma \leftarrow get\text{-}Store$$
$$\kappa \sigma \leftarrow get\text{-}KStore$$
$$l*_0 \leftarrow R_0(\rho, e)$$
$$\kappa l_0 \leftarrow get\text{-}KAddr$$
$$\texttt{let } l*' := \mu(\theta).l *_0 \cup \theta \cup R[\sigma](\theta)$$
$$\texttt{let } \kappa l*' := \mu(\kappa\theta).\{\kappa l_0\} \cup \kappa\theta \cup KR[\kappa\sigma](\kappa\theta)$$
$$put\text{-}Store(\{l \mapsto \sigma(l) \mid l \in l*'\})$$
$$put\text{-}KStore(\{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in \kappa l*'\})$$

where R_0 is defined as before and *R*, *KR* and *R-Clo* are defined:

$$R\colon \textit{Store} \to \mathcal{P}(\textit{Addr}) \to \mathcal{P}(\textit{Addr})$$
$$R[\sigma](\theta) \coloneqq \{l' \mid l' \in \textit{R-Clo}(c)\ ;\ c \in \textit{clo-E}(v)\ ;\ v \in \sigma(l)\ ;\ l \in \theta\}$$
$$\textit{R-Clo}\colon \textit{Clo} \to \mathcal{P}(\textit{Addr})$$
$$\textit{R-Clo}(\langle[\lambda](x).e, \rho\rangle) \coloneqq \{\rho(x) \mid x \in \textit{FV}([\lambda](x).e)\}$$
$$\textit{KR}\colon \textit{KStore} \to \mathcal{P}(\textit{KAddr}) \to \mathcal{P}(\textit{KAddr})$$
$$\textit{KR}[\sigma](\kappa\theta) \coloneqq \{\pi_2(fr) \mid fr \in \kappa\sigma(\kappa l)\ ;\ \kappa l \in \theta\}$$

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 $\alpha \to M(\beta)$ form a Galois connection with a transition system $\Sigma \to \Sigma$.

There is one last parameter to our development: a connection between our monadic interpreter and a state space transition system. We state this connection formally as a Galois connection $(\Sigma \to \Sigma) \xleftrightarrow[\alpha]{\gamma} (\textit{Exp} \to M(\textit{Exp}))$. 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*. For example, given concrete and abstract versions of *Val*, we carry $\mathbf{Val} \xleftrightarrow[\alpha]{\gamma} \widehat{\mathbf{Val}}$ through the Galois connection to establish $\Sigma \xleftrightarrow[\alpha]{\gamma} \widehat{\Sigma}$.

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

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

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

6. Recovering Interpreters

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**, a powerset of values:

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

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

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

and the concrete δ you would expect:

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

Proposition 1. **Val** *satisfies the abstract domain laws from section X.*

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

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

and *tick* is just a cons operator:

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

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

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

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

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

State effects merely return singleton sets:

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

Nondeterminism effects are implemented with set union:

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

Proposition 2. **M** *satisfies monad, state, and nondeterminism laws.*

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

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

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

$$\gamma\colon (Exp \to \mathbf{M}(Exp)) \to (\boldsymbol{\Sigma} \to \boldsymbol{\Sigma})$$
$$\gamma(f)(e\psi*) := \{(e,\psi') \mid (e,\psi') \in f(e)(\psi) \,;\, (e,\psi) \in e\psi*\}$$
$$\alpha\colon (\boldsymbol{\Sigma} \to \boldsymbol{\Sigma}) \to (Exp \to \mathbf{M}(Exp))$$
$$\alpha(f)(e)(\psi) := f(\{(e,\psi)\})$$

The injection $\varsigma_0^{\mathbf{M}}$ for a program e_0 is:

$$\varsigma_0 := \{\langle e, \bot, \bot, , \bot, \rangle\}$$

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.

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

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

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

Introduction and elmination for *Clo* is identical to the concrete domain.

The abstract δ operator is defined:

$$\delta\colon IOp \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Val}} \to \widehat{\mathbf{Val}}$$
$$\delta(+, v_1, v_2) :=$$
$$\{i \mid 0 \in v_1 \wedge i \in v_2\}$$
$$\cup \{i \mid i \in v_1 \wedge 0 \in v_2\}$$
$$\cup \{+ \mid + \in v_1 \wedge + \in v_2\}$$
$$\cup \{- \mid - \in v_1 \wedge - \in v_2\}$$
$$\cup \{-, 0, + \mid + \in v_1 \wedge - \in v_2\}$$
$$\cup \{-, 0, + \mid - \in v_1 \wedge + \in v_2\}$$

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

Proposition 4. $\widehat{\mathbf{Val}}$ *satisfies the abstract domain laws from section X.*

Proposition 5. Val $\xrightarrow[\alpha]{\gamma}$ $\widehat{\mathbf{Val}}$ *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:

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

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

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

Proposition 6. Time $\xrightarrow[\alpha]{\gamma}$ $\widehat{\mathbf{Time}}$ *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 $\widehat{\mathbf{Time}}$.

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

The resulting state space $\widehat{\boldsymbol{\Sigma}}$ 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 $\widehat{\mathbf{M}}$: $\widehat{\mathbf{M}}^{fi}$. To do this we pull $\widehat{\mathbf{Store}}$ out of the powerset, exploiting its join-semilattice structure:

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

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

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

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

$$return\colon \forall \alpha, \alpha \to \widehat{\mathbf{M}}(\alpha)$$
$$return(a)(\psi, \sigma) := (\{a, \psi\}, \sigma)$$

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

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

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

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

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

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

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

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

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

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

Proposition 8. *There exists Galois connection* $\Sigma \xrightleftharpoons[\alpha_1]{\gamma_1} \widehat{\Sigma} \xrightleftharpoons[\alpha_2]{\gamma_2} \widehat{\Sigma}^{fi}$ *and* $\alpha_1 \circ C\gamma(step) \circ \gamma_1 \sqsubseteq A\gamma(step) \sqsubseteq \gamma_2 \circ A\gamma^{fi}(step) \circ \alpha_2.$

The first Galois connection $\Sigma \xrightleftharpoons[\alpha_1]{\gamma_1} \widehat{\Sigma}$ is justified by the Galois connections between $\mathbf{Val} \xrightleftharpoons[\alpha]{\gamma} \widehat{\mathbf{Val}}$ and $\mathbf{Time} \xrightleftharpoons[\alpha]{\gamma} \widehat{\mathbf{Time}}$. The second Galois connection $\widehat{\Sigma} \xrightleftharpoons[\alpha_2]{\gamma_2} \widehat{\Sigma}^{fi}$ 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 $\widehat{\Sigma}$ and $\widehat{\Sigma}^{fi}$. 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 interprer: 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, $S_t[s]$:

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

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 $\mathtt{do}_m$ and $\leftarrow_m$ for clarity.

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

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

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

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

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

$$get \colon S_t[s](m)(s)$$
$$get(s) \coloneqq return_m(s, s)$$
$$put \colon s \to S_t[s](m)(1)$$
$$put(s')(s) \coloneqq return_m(1, s')$$

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:

$$\mathcal{P}_t \colon (Type \to Type) \to (Type \to Type)$$
$$\mathcal{P}_t(m)(\alpha) \coloneqq m(\mathcal{P}(\alpha))$$

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

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

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 $\mathcal{P}_t$:

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

Proposition 10. get *and* put *satisfy the state monad laws.*

The proof is by simpl calculation.

Finally, our nondeterminism monad transformer expses nondeterminism effects as a trivial applciation of the underlying monad's join-semilattice functorality:

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

Proposition 11. mzero *and* $\langle+\rangle$ *satisfy the nondterminism 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:

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

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:

$$mstep - \gamma\colon \forall\alpha\beta m, (\alpha \to S_t[s](m)(\beta)) \to (\Sigma_m(\alpha \times s) \to \Sigma_m(\beta \times s))$$
$$mstep - \gamma(f) \coloneqq mstep_m - \gamma(\lambda(a,s).f(a)(s))$$

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

$$mstep_1 - \gamma\colon \forall\alpha\beta m, (\alpha \to \mathcal{P}_t(m)(\beta)) \to (\Sigma_m(\mathcal{P}(\alpha)) \to \Sigma_m(\mathcal{P}(\beta)))$$
$$mstep_1 - \gamma(f) \coloneqq mstep_m - \gamma(\lambda(\{x_1..x_n\}).f(x_1)\langle+\rangle .. \langle+\rangle f(x_n))$$

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

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

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

$$commuteP - \gamma\colon \forall\alpha, \Sigma_m(\mathcal{P}(\alpha) \times s) \to \mathcal{P}(\Sigma_m(\alpha \times s))$$
$$commuteP - \gamma \coloneqq commuteP_m \circ map(\lambda(\{\alpha_1..\alpha_n\},s).\{(\alpha_1,s)..(\alpha_n,s)\})$$

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 $mstep_{\mathcal{P}_t1}$ are really isomorphism transformers, and $mstep_{\mathcal{P}_t2}$ 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 $mstep_m - \alpha$ to be functorial (i.e., homomorphic).

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

The proof is by consequence of *commuteP*.

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

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

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

which yields a path-sensitive flow-sensitive analysis,

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

which yeilds a path-insensitive flow-sensitive analysis, and

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

which yields a path-insensitive flow-insensitive analysis. Furthermore, the Galois connections for our interpreter instantiated to each state space Σ is justified fully by construction.