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 shows the full definition of a highly parameterized monadic interpreter. Section X 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 metatheory framework built on monad transformers.

2. Semantics

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

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

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 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 IF is a standard CESK machine augmented with a separate store for continuation values:

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

The semantics of atomic terms is given denotationally with the denotation function $A[\![\_, \_, \_]\!]$:

$$
\begin{array}{rl}
A[\![\_, \_, \_]\!] & \in \mathit{Env} \times \mathit{Store} \times \mathit{Atom} \rightharpoonup \mathit{Val} \\
A[\![\rho, \sigma, i]\!] & := i \\
A[\![\rho, \sigma, x]\!] & := \sigma(\rho(x)) \\
A[\![\rho, \sigma, \underline{\boldsymbol{\lambda}}(x).e]\!] & := \langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle \\
[\![\_, \_, \_]\!] & \in \mathit{IOp} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
[\![+, i_1, i_2]\!] & := i_1 + i_2 \\
[\![-, i_1, i_2]\!] & := i_1 - i_2
\end{array}
$$

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

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

Our abstract 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:

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

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 adn *R-Val* for computing addresses reachable from addresses.

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

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

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

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

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

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

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

3. Monadic Interpreter

In this section we design a monadic interpreter for the IF language which is also parameterizedin AAM[CITE] style. 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.

3.1 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.

3.1.1 Monad Properties

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

$$\mathit{bind} : \forall\alpha\beta, M(\alpha) \to (\alpha \to M(\beta)) \to M(\beta)$$

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

$$\mathit{return} : \forall\alpha, \alpha \to M(\alpha)$$

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

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

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. 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:

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

and

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

are both just sugar for

$$\mathit{bind}(m)(k)$$

3.1.2 Monad State Properties

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

$$\textit{get-Env} : M(\textit{Env})$$
$$\textit{put-Env} : \textit{Env} \to M(1)$$

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

$$\textit{put-put} : \textit{put-Env}(s_1)\ ;\ \textit{put-Env}(s_2) = \textit{put-Env}(s_2)$$
$$\textit{put-get} : \textit{put-Env}(s)\ ;\ \textit{get-Env} = \textit{return}(s)$$
$$\textit{get-put} : s \leftarrow \textit{get-Env}\ ;\ \textit{put-Env}(s) = \textit{return}(1)$$
$$\textit{get-get} : s_1 \leftarrow \textit{get-Env}\ ;\ s_2 \leftarrow \textit{get-Env}\ ;\ k(s_1, s_2) = s \leftarrow \textit{get-Env}\ ;\ k(s, s)$$

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

3.1.3 Monad Nondeterminism Properties

Nondeterminism is achieved through operators $\langle 0 \rangle$ and $\langle + \rangle$:

$$\langle 0 \rangle : \forall \alpha, M(\alpha)$$
$$\_\ \langle + \rangle\ \_ : \forall \alpha, M(\alpha) \times M(\alpha) \to M(\alpha)$$

We use the nondeterminism laws to reason about nondeterminism effects:

$$\bot\texttt{-zero}_1 : \textit{bind}(\langle 0 \rangle)(k) = \langle 0 \rangle$$
$$\bot\texttt{-zero}_2 : \textit{bind}(m)((a).\langle 0 \rangle) = \langle 0 \rangle$$
$$\bot\texttt{-unit}_1 : \langle 0 \rangle\ \langle + \rangle\ m = m$$
$$\bot\texttt{-unit}_2 : m\ \langle + \rangle\ \langle 0 \rangle = m$$
$$+\texttt{-assoc} : m_1\ \langle + \rangle\ (m_2\ \langle + \rangle\ m_3) = (m_1\ \langle + \rangle\ m_2)\ \langle + \rangle\ m_3$$
$$+\texttt{-comm} : m_1\ \langle + \rangle\ m_2 = m_2\ \langle + \rangle\ m_1$$
$$+\texttt{-dist} : \textit{bind}(m_1\ \langle + \rangle\ m_2)(k) = \textit{bind}(m_1)(k)\ \langle + \rangle\ \textit{bind}(m_2)(k)$$

3.2 The Abstract Domain

The abstract domain is encapsulated by the *Val* type in the semantics. To parameterize over it, we leave *Val* opaque but require it support various operations. There is a constraint on *Val* its-self: it must be a join-semilattice:

$$\bot : \textit{Val}$$
$$\_ \sqcup \_ : \textit{Val} \times \textit{Val} \to \textit{Val}$$

We require *Val* to be a join-semilattice so they can be merged in the *Store*.

The interface for integers consists of introduction and elimiation rules:

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

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

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

Additionally we must abstract closures:

$$\textit{clo-I} : \textit{Clo} \to \textit{Val}$$
$$\textit{clo-E} : \textit{Val} \to \mathcal{P}(\textit{Clo})$$

which follow similar laws:

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

The denotation for primitive operations must also be opaque:

$$[\![\_, \_, \_]\!] : \textit{IOp} \times \textit{Val} \times \textit{Val} \to \textit{Val}$$

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

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

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.

3.3 Abstract Time

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

$$\textit{tick} : \textit{Exp} \times \textit{KAddr} \times \textit{Time} \to \textit{Time}$$

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 $\textit{Exp} \times \textit{KAddr}$. Also in AAM is the opaque function $alloc : \textit{Var} \times \textit{Time} \to \textit{Addr}$. 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.

3.4 Interpreter Definition

We now present a generic monadic interpreter for 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*:

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

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 $\mathcal{P}(\mathit{Frame} \times \mathit{KAddr})$ 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 $\langle 0 \rangle$, which we name $\uparrow_p$:

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

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

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

Finally, we introduce helper functions for manipulating stack frames:

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

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.

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

The step function is a monadic computation from

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

We also implement abstract garbage collection monadically:

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

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

$$
\begin{array}{l}
R : \mathit{Store} \to \mathcal{P}(\mathit{Addr}) \to \mathcal{P}(\mathit{Addr}) \\
R[\sigma](\theta) := \{l' \mid l' \in R - Clo(c)\ ;\ c \in \mathit{clo\text{-}E}(v)\ ;\ v \in \sigma(l)\ ;\ l \in \theta\} \\
R - Clo : \mathit{Clo} \to \mathcal{P}(\mathit{Addr}) \\
R - Clo(\langle \underline{\boldsymbol{\lambda}}(x).e, \rho \rangle) := \{\rho(x) \mid x \in \mathit{FV}(\underline{\boldsymbol{\lambda}}(x).e)\} \\
KR : \mathit{KStore} \to \mathcal{P}(\mathit{KAddr}) \to \mathcal{P}(\mathit{KAddr}) \\
KR[\sigma](\kappa\theta) := \{\pi_2(fr) \mid fr \in \kappa\sigma(\kappa l)\ ;\ \kappa l \in \theta\}
\end{array}
$$

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 $(\Sigma \to \Sigma)\alpha(\mathit{Exp} \to \mathit{M}(\mathit{Exp}))$. 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 . Second, this Galois connection serves to *transport other Galois connections*. For example, given concrete and abstract versions of *Val*, we carry $\mathit{Val}\alpha\widehat{\mathit{Val}}$ through the Galois connection to establish $C\Sigma\alpha A\Sigma$.

A collecting-semantics execution of our interpreter is defined as:

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

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

4. Recovering Concrete and Abstract Interpreters

To recover a concrete interpreter we instantiate *M* to a path-sensitive monad: M^{ps}. The path sensitive monad is a simple powerset of products:

$$
\begin{array}{l}
\in\ ^{ps} := \mathit{Env} \times \mathit{Store} \times \mathit{KAddr} \times \mathit{KStore} \times \mathit{Time} \\
m \in M^{ps}(\alpha) := {}^{ps} \to \mathcal{P}(\alpha \times {}^{ps})
\end{array}
$$

Monadic operators bind^{ps} and return^{ps} are defined to encapsulate both state-passing and set-flattening:

$$
\begin{array}{l}
\mathit{bind}^{ps} : \forall\alpha, M^{ps}(\alpha) \to (\alpha \to M^{ps}(\beta)) \to M^{ps}(\beta) \\
\mathit{bind}^{ps}(m)(f)() := \{(y,'') \mid (y,'') \in f(a)(')\ ;\ (a,') \in m()\} \\
\mathit{return}^{ps} : \forall\alpha, \alpha \to M^{ps}(\alpha) \\
\mathit{return}^{ps}(a)() := \{(a,)\}
\end{array}
$$

State effects merely return singleton sets:

$$
\begin{array}{l}
\mathit{get\text{-}Env}^{ps} : M^{ps}(\mathit{Env}) \\
\mathit{get\text{-}Env}^{ps}(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(\rho, \langle \rho, \sigma, \kappa, \tau \rangle)\} \\
\mathit{put\text{-}Env}^{ps} : \mathit{Env} \to \mathcal{P}(1) \\
\mathit{put\text{-}Env}^{ps}(\rho')(\langle \rho, \sigma, \kappa, \tau \rangle) := \{(1, \langle \rho', \sigma, \kappa, \tau \rangle)\}
\end{array}
$$

Nondeterminism effects are implemented with set union:

$$
\begin{array}{l}
\langle 0 \rangle^{ps} : \forall\alpha, M^{ps}(\alpha) \\
\langle 0 \rangle^{ps}() := \{\} \\
\_\ \langle + \rangle\ ^{ps}\_ : \forall\alpha, M^{ps}(\alpha) \times M^{ps}(\alpha) \to M^{ps}(\alpha) \\
(m_1\ \langle + \rangle\ ^{ps} m_2)() := m_1() \cup m_2()
\end{array}
$$

Proposition: M satisfies monad, state, and nondeterminism laws.

For the value space *Val* we use a powerset of semantic values *Val*:

$$v \in \mathit{Val} := \mathcal{P}(\mathit{Val})$$

with introduction and elimination rules:

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

and to manipulate abstract values:

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

Abstract time and addresses are program contours in the concrete space:

$$\begin{aligned}
\tau \in \mathit{Time} &:= (\mathit{Exp} \times \mathit{KAddr})*\\
l \in \mathit{Addr} &:= \mathit{Var} \times \mathit{Time}\\
\kappa l \in \mathit{KAddr} &:= \mathit{Time}
\end{aligned}$$

Operators $alloc$ and $\kappa alloc$ are merely identity functions, and *tick* is just a cons operator.

Finally, we must establish a Galois connection between $\mathit{Exp} \to M^{ps}(\mathit{Exp})$ and $\Sigma \to \Sigma$ for some Σ. The state space Σ depends only on the monad M^{ps} and is independent of the choice for *Val*, Addr or Time. For the path sensitive monad M^{ps} , Σ^{ps} is defined:

$$\Sigma^{ps} := \mathcal{P}(\mathit{Exp} \times {}^{ps})$$

and the Galois connection is:

$$\begin{aligned}
&{}^{ps} : (\mathit{Exp} \to M^{ps}(\mathit{Exp})) \to \Sigma^{ps} \to \Sigma^{ps}\\
&{}^{ps}(f)(e*) := \{(e,') \mid (e,') \in f(e)() \; ; \; (e,) \in e*\}\\
&\alpha^{ps} : (\Sigma^{ps} \to \Sigma^{ps}) \to \mathit{Exp} \to M^{ps}(\mathit{Exp})\\
&\alpha^{ps}(f)(e)() := f(\{(e,)\})
\end{aligned}$$

Proposition: ^{ps} and α^{ps} form an isomorphism.

This implies Galois connnection.

The injection ς_0^{ps} for a program e is:

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

To arrive at an abstract interpreter we seek a finite state space. First we abstract the value space *Val* as $\widehat{\mathit{Val}}$, which only tracks integer parity:

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

Introduction and elimination functions are defined:

$$\begin{aligned}
&\mathit{int\text{-}I} : \mathbb{Z} \to \widehat{\mathit{Val}}\\
&\mathit{int\text{-}I}(i) := -\,\texttt{if}\; i < 0\\
&\qquad\qquad\qquad\qquad [0]\,\texttt{if}\; i = 0\\
&\qquad\qquad\qquad\qquad +\,\texttt{if}\; i > 0\\
&\mathit{int\text{-}if0\text{-}E} : \widehat{\mathit{Val}} \to \mathcal{P}(\mathit{Bool})\\
&\mathit{int\text{-}if0\text{-}E}(v) := \{\mathit{true} \mid 0 \in v\} \cup \{\mathit{false} \mid - \in v+ \in v\}
\end{aligned}$$

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

The abstract operator is defined:

$$\begin{aligned}
&A : \mathit{IOp} \times \widehat{\mathit{Val}} \times \widehat{\mathit{Val}} \to \widehat{\mathit{Val}}\\
&A(+, v_1, v_2) := \{p \mid [0] \in v_1 p \in v_2\}\\
&\qquad\qquad \cup \{p \mid p \in v_1[0] \in v_2\}\\
&\qquad\qquad \cup \{+ \mid + \in v_1+ \in v_2\}\\
&\qquad\qquad \cup \{- \mid - \in v_1- \in v_2\}\\
&\qquad\qquad \cup \{-, [0], + \mid + \in v_1- \in v_2\}\\
&\qquad\qquad \cup \{-, [0[, + \mid - \in v_1+ \in v_2\}
\end{aligned}$$

Next we abstract *Time* to the finite domain of a k-truncated list of execution contexts:

$$\mathit{Time} := (\mathit{Exp} \times \mathit{KAddr}) *$$

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

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

After substituting abstract versions for *Val* and *Time*, the following state space for Σ^{ps} becomes finite:

$$\mathcal{P}(\mathit{Exp} \times \mathit{AEnv} \times \mathit{AStore} \times \mathit{AKAddr} \times \mathit{AKStore} \times \mathit{ATime})$$

and the least-fixed-point iteration of the collecting semantics provides a sound and computable analysis.

5. Varying Path and Flow Sensitivity

We are able to recover a flow-insensitive interpreter through a new definition for *M*: M^{fi}. To do this we pull *Store* out of the powerset and use its join-semilattice structure:

$$\begin{aligned}
{}^{fi} &:= \mathit{Env} \times \mathit{KAddr} \times \mathit{KStore} \times \mathit{Time}\\
M^{fi}(\alpha) &:= {}^{fi} \times \mathit{Store} \times \mathcal{P}(\alpha \times {}^{fi}) \times \mathit{Store}
\end{aligned}$$

The monad operator $bind^{fi}$ must merge multiple stores back to one:

$$bind^{fi} : \forall\alpha\beta, M^{fi}(\alpha) \to (\alpha \to M^{fi}(\beta)) \to M^{fi}(\beta)$$
$$bind^{fi}(m)(f)(,\sigma) := (\{bs_{11}..bs_{n1}..bs_{nm}\}, \sigma_1 \sqcup .. \sqcup \sigma_n)$$
where
$$(\{(a_1,_1)..(a_n,_n)\}, \sigma') := m(,\sigma)$$
$$(\{b_{i1}..b_{im}\}, \sigma_i) := f(a_i)(_i, \sigma')$$

The unit for $bind^{fi}$:

$$return^{fi} : \forall\alpha, \alpha \to M^{fi}(\alpha)$$
$$return^{fi}(a)(,\sigma) := (\{a,\}, \sigma)$$

State effects *get-Env* and *put-Env*:

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

State effects *get-Store* and *put-Store*:

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

Nondeterminism operations:

$$\langle 0\rangle^{fi} : \forall\alpha, M(\alpha)$$
$$\langle 0\rangle^{fi}(,\sigma) := (\{\}, \bot)$$
$$\_ \ \langle+\rangle \ \_ : \forall\alpha, M(\alpha) \times M(\alpha) \to M\alpha$$
$$(m_1 \ \langle+\rangle \ m_2)(,\sigma) := (a *_1 \cup a *_2, \sigma_1 \sqcup \sigma_2)$$
where $(a*_i, \sigma_i) := m_i(,\sigma)$

Finally, the Galois connection for relating M^{fi} to a state space transition over Σ^{fi}:

$$\Sigma^{fi} := \mathcal{P}(Exp \times {}^{fi}) \times Store$$
$${}^{fi} : (Exp \to M^{fi}(Exp)) \to (\Sigma^{fi} \to \Sigma^{fi})$$
$${}^{fi}(f)(e*,\sigma) := (\{e_{11}..e_{n1}..e_{nm}\}, \sigma_1 \sqcup .. \sqcup \sigma_n)$$
where $\{(e_1,_1)..(e_n,_n)\} := e*$
$$(\{e_{i1}..e_{im}\}, \sigma_i) := f(e_i)(_i, \sigma)$$
$$\alpha^{fi} : (\Sigma^{fi} \to \Sigma^{fi}) \to (Exp \to M^{fi}(Exp))$$
$$\alpha^{fi}(f)(e)(,\sigma) := f(\{(e,)\}, \sigma)$$

Proposition: ^{fi} *and* α^{fi} *form an isomorphism.*

Like the concrete ^{fi} and α^{fi}, this implies Galois connection.

Proposition: $\mathrm{M}^{ps}\alpha\mathrm{M}^{fi}$.

This demonstrates that path sensitivity is more precise than flow insensitivity in a formal, language-independent setting.

We leave out the explicit definition for the flow-sensitive monad M^{fs}. However, we will recover it through the compositional framework in Section [X][A Compositional Framework] using monad transformers.

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.

6. A Compositional Monadic Framework

In our framework thus far, any modification to the interpreter requires redesigning the monad M. However, we want to avoid reconstructing complicated monads for our interpreters. Even more, we want to avoid reconstructing *proofs* about monads for our interpreters. Toward this goal we introduce a compositional framework for constructing monads using a restricted class of monad transformer.

There are two types of monadic effects used in the monadic interprer: state and nondeterminism. There is a monad transformer for adding state effects to existing monads, called the state monad tranformer:

$$S[\_] : (Type \to Type) \to (Type \to Type)$$
$$S[s](m)(\alpha) := s \to m(\alpha \times s)$$

Monadic actions *bind* and *return* (and their laws) use the underlying monad:

$$bind : \forall\alpha\beta, S[s](m)(\alpha) \to (\alpha \to S[s](m)(\beta)) \to S[s](m)(\beta)$$
$$bind(m)(f)(s) := \texttt{do}$$
$$(x, s') \leftarrow^m m(s)$$
$$f(x)(s')$$
$$return : \forall\alpha m, \alpha \to S[s](m)(\alpha)$$
$$return\,(x)(s) := return^m(x, s)$$

State actions get and put expose the cell of state while interacting with the underlying monad m:

$$get : S[s](m)(s)$$
$$get(s) := return^m(s, s)$$
$$put : s \to S[s](m)(1)$$
$$put(s')(s) := return^m(1, s')$$

and the state monad transformer is able to transport nondeterminism effects from the underlying monad:

$$\langle 0\rangle : \forall\alpha, S[s](m)(\alpha)$$
$$\langle 0\rangle(s) := \langle 0\rangle^m$$
$$\_ \ \langle+\rangle \ \_ : \forall\alpha, S[s](m)(\alpha) x S[s](m)(\alpha) \to S[s](m)(\alpha)$$
$$(m_1 \ \langle+\rangle \ m_2)(s) := m_1(s) \ \langle+\rangle \ {}^m m_2(s)$$

The state monad transformer was introduced by Mark P. Jones in [X].

We develop a new monad transformer for nondeterminism which can compose with state in both directions.

$$\mathcal{P} : (Type \to Type) \to (Type \to Type)$$
$$\mathcal{P}(m)(\alpha) := m(\mathcal{P}(\alpha))$$

Monadic actions *bind* and *return* require that the underlying monad be a join-semilattice functor:

$$\begin{aligned}
&\mathit{bind} : \forall \alpha\beta, \mathcal{P}(m)(\alpha) \to (\alpha \to \mathcal{P}(m)(\beta)) \to \mathcal{P}(m)(\beta)\\
&\mathit{bind}(m)(f) := \texttt{do}\\
&\quad \{x_1..x_n\} \leftarrow^m m\\
&\quad f(x_1) \sqcup^m .. \sqcup^m f(x_n)\\
&\mathit{return} : \forall \alpha, \alpha \to \mathcal{P}(m)(\alpha)\\
&\mathit{return}\,(x) := \mathit{return}^m(\{x\})
\end{aligned}$$

Nondterminism actions $\langle 0 \rangle^m and +$ interact with the join-semilattice functorality of the underlying monad m:

$$\begin{aligned}
&\langle 0 \rangle : \forall \alpha, \mathcal{P}(m)(\alpha)\\
&\langle 0 \rangle := \bot^m\\
&_ \ \langle + \rangle \ _ : \forall \alpha, \mathcal{P}(m)(\alpha) x \mathcal{P}(m)(\alpha) \to \mathcal{P}(m)(\alpha)\\
&m_1 \ \langle + \rangle \ m_2 := m_1 \sqcup^m m_2
\end{aligned}$$

and the nondeterminism monad transformer is able to transport state effects from the underlying monad:

$$\begin{aligned}
&get : \mathcal{P}(m)(s)\\
&get = map((s).\{s\})(get^m)\\
&put : s \to \mathcal{P}(m)(s)\\
&put(s) = map((1).\{1\})(put^m(s))
\end{aligned}$$

Proposition: $\mathcal{P}$ is a transformer for monads which are also join semi-lattice functors.

Our correctness framework requires that monadic actions in *M* map to state space transitions in Σ. We establish this property in addition to monadic actions and effects for state and nondeterminism monad transformers. We call this property $MonadStep$, where monadic acations in *M* admit a Galois connection to transitions in Σ:

$$mstep : \forall \alpha\beta, (\alpha \to \mathit{M}(\beta))\alpha(\Sigma(\alpha) \to \Sigma(\beta))$$

We now show that the monad transformers for state and nondeterminism transport this property in addition to monadic operations.

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

$$\begin{aligned}
&mstep - \ : \forall \alpha\beta m, (\alpha \to S[s](m)(\beta)) \to (\Sigma^m(\alpha \times s) \to \Sigma^m(\beta \times s))\\
&mstep - (f) := mstep^m - ((a, s).f(a)(s))
\end{aligned}$$

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

$$\begin{aligned}
&mstep_1 - \ : \forall \alpha\beta m, (\alpha \to \mathcal{P}(m)(\beta)) \to (\Sigma^m(\mathcal{P}(\alpha)) \to \Sigma^m(\mathcal{P}(\beta)))\\
&mstep_1 - (f) := mstep^m - (((\{x_1..x_n\}).f(x_1) \ \langle + \rangle \ .. \ \langle + \rangle \ f(x_n))
\end{aligned}$$

and one where Σ is $P\Sigma^m$:

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

The operation $computeP$ must be defined for the underlying Σ^m. This property is true for the identiy monad, and is preserved by $S[s]$ when Σ^m is also a functor:

$$\begin{aligned}
&commuteP - \ : \forall \alpha, \Sigma^m(\mathcal{P}(\alpha) \times s) \to \mathcal{P}(\Sigma^m(\alpha \times s))\\
&commuteP - \ := commuteP^m map((\{\alpha_1..\alpha_n\}, s).\{(\alpha_1, s)..(\alpha_n, s)\})
\end{aligned}$$

The side of commuteP is the only Galois connection mapping that loses information in the α direction. Therefore, $mstep$ and $mstep_1$ are really isomorphism transformers, and $mstep_2$ is the only Galois connection transformer.

[QUESTION: should I give the definitions for the α maps here? -DD]

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

We can now build monad transformer stacks from combinations of $S[s]$, FI and FS 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.
- Galois connections between states s_1 and s_2 are transported along the Galois connection between $(\alpha \to S[s_1](m)(\beta))\alpha(\Sigma[s_1](\alpha) \to \Sigma[s_1](\beta))$ and $(\alpha \to S[s_2](m)(\beta))\alpha(\Sigma[s$ $\Sigma[s_2](\beta))$ resulting in $(\Sigma[s_1](\alpha) \to \Sigma[s_1](\beta))\alpha\beta(\Sigma[s_2](\alpha) \to$ $\Sigma[s_2](\beta))$.

We can now instantiate our interpreter to the following monad stacks.

- $S[Env]S[Store]S[KAddr]S[KStore]S[Time]FS$
  - This yields a path-sensitive flow-sensitive analysis.
- $S[Env]S[KAddr]S[KStore]S[Time]FSS[Store]$
  - This yeilds a path-insensitive flow-sensitive analysis.
- $S[Env]S[KAddr]S[KStore]S[Time]FIS[Store]$
  - This yields a path-insensitive flow-insensitive analysis.

Furthermore, the final Galois connection for each state space is justified from individual Galois connections between state space components.

A. Typesetting

$$
\begin{aligned}
A\_ &\in \mathit{Atom} \to M(\mathit{Val}) \\
Ai &:= \mathit{return}(\textit{int-I}(i)) \\
Ax &:= \mathit{do} \\
&\quad \rho \leftarrow \textit{get-Env} \\
&\quad \sigma \leftarrow \textit{get-Store} \\
&\quad \ell \leftarrow {}_p(\rho(x)) \\
&\quad \mathit{return}(\sigma(x)) \\
A\lambda(x).e &:= \mathit{do} \\
&\quad \rho \leftarrow \textit{get-Env} \\
&\quad \mathit{return}(\textit{clo-I}(\langle \lambda(x).e, \rho \rangle))
\end{aligned}
$$