

Modular Metatheory for Abstract Interpreters

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.
- A new monad transformer for nondeterminism.
- An isolated understanding of flow and path sensitivity for analysis as mere variations in the order of monad transformer composition.

1.2 Outline

We will demonstrate our framework by example, walking the reader through the design and implementation of a family of an abstract interpreter. Section 2 gives the concrete semantics for a small functional language. Section 3 shows the full definition of a concrete monadic interpreter. Section 4 [A Compositional Monadic Framework] shows our compositional meta-theory framework built on monad transformers.

2. Semantics

Our language of study is λIF :

```
i  ∈ ℤ
x  ∈ Var
a  ∈ Atom ::= i | x | λ(x).e
iop ∈ IOp  ::= + | -
op  ∈ Op   ::= iop | @
e  ∈ Exp  ::= a | e op e | if0(e){e}{e}
```

[Copyright notice will appear here once 'preprint' option is removed.]

(The operator @ is syntax for function application; We define op as a single syntactic class for all operators to simplify presentation.) We begin with a concrete semantics for λIF which makes allocation explicit. Using an allocation semantics has several consequences for the abstract semantics:

- Call-site sensitivity can be recovered through choice of abstract time and address.
- Abstract garbage collection can be performed for unreachable abstract values.
- Widening techniques can be applied to the store.

The concrete semantics for λIF :

```

 $\tau \in \text{Time} \quad := \mathbb{Z}$ 
 $l \in \text{Addr} \quad := \text{Var} \times \mathbb{Z}$ 
 $\rho \in \text{Env} \quad := \text{Var} \rightarrow \text{Addr}$ 
 $\sigma \in \text{Store} := \text{Addr} \rightarrow \text{Val}$ 
 $f \in \text{Frame} ::= [\Box \text{ op } e] \mid [v \text{ op } \Box] \mid [\text{if}\theta(\Box)\{e\}\{e\}]$ 
 $\kappa \in \text{Kon} \quad := \text{Frame}^*$ 
 $c \in \text{Clo} \quad := (\lambda(x).e, \rho)$ 
 $v \in \text{Val} \quad := i \mid c$ 
 $\zeta \in \Sigma \quad := \text{Exp} \times \text{Env} \times \text{Store} \times \text{Kon}$ 

 $\text{alloc} \in \text{Var} \times \text{Time} \rightarrow \text{Addr}$ 
 $\text{alloc}(x, \tau) := \langle x, \tau \rangle$ 

 $\text{tick} \in \text{Time} \rightarrow \text{Time}$ 
 $\text{tick}(\tau) := \tau + 1$ 

 $A[\_, \_, \_] \in \text{Env} \times \text{Store} \times \text{Atom} \rightarrow \text{Val}$ 
 $A[\rho, \sigma, i] := i$ 
 $A[\rho, \sigma, x] := \sigma(\rho(x))$ 
 $A[\rho, \sigma, \lambda(x).e] := (\lambda(x).e, \rho)$ 

 $\delta[\_, \_, \_] \in \text{IOp} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ 
 $\delta[+, i_1, i_2] := i_1 + i_2$ 
 $\delta[-, i_1, i_2] := i_1 - i_2$ 

 $\_ \rightsquigarrow \_ \in \text{P}(\Sigma \times \Sigma)$ 
 $\langle e_1 \text{ op } e_2, \rho, \sigma, \kappa, \tau \rangle \rightsquigarrow \langle e_1, \rho, \sigma, [\Box \text{ op } e_2]::\kappa, \text{tick}(\tau) \rangle$ 
 $\langle a, \rho, \sigma, [\Box \text{ op } e]::\kappa, \tau \rangle \rightsquigarrow \langle e, \rho, \sigma, [v \text{ op } \Box]::\kappa, \text{tick}(\tau) \rangle$ 
  where  $v = A[\rho, \sigma, a]$ 
 $\langle a, \rho, \sigma, [v_1 @ \Box]::\kappa, \tau \rangle \rightsquigarrow \langle e, \rho'[\lambda \mapsto l], \sigma[l \mapsto v_2], \kappa, \text{tick}(\tau) \rangle$ 
  where  $\langle \lambda(x).e, \rho' \rangle := v_1$ 
 $v_2 := A[\rho, \sigma, a]$ 
 $l := \text{alloc}(x, \tau)$ 

 $\langle i_2, \rho, \sigma, [i_1 \text{ iop } \Box]::\kappa, \tau \rangle \rightsquigarrow \langle i, \rho, \sigma, \kappa, \text{tick}(\tau) \rangle$ 
  where  $i := \delta[\text{iop}, i_1, i_2]$ 
 $\langle i, \rho, \sigma, [\text{if}\theta(\Box)\{e_1\}\{e_2\}]::\kappa, \tau \rangle \rightsquigarrow \langle e, \rho, \sigma, \kappa, \text{tick}(\tau) \rangle$ 
  where  $e := e_1$  if  $i = \theta$ 
 $e_2$  otherwise

```

We also wish to employ abstract garbage collection, which adheres to the following specification:

```

 $\_ \rightsquigarrow \_ \in \text{P}(\Sigma \times \Sigma)$ 
 $\zeta \rightsquigarrow \zeta'$  where  $\zeta \rightsquigarrow \zeta'$ 
 $\langle e, \rho, \sigma, \kappa, \tau \rangle \rightsquigarrow \langle e, \rho, \{\lambda \mapsto \sigma(l) \mid l \in R[\rho, \sigma](e, \kappa)\}, \kappa, \tau \rangle$ 

 $R[\_, \_] \in \text{Env} \times \text{Store} \rightarrow \text{Exp} \times \text{Kon} \rightarrow \text{P}(\text{Addr})$ 
 $R[\rho, \sigma](e, \kappa) := \mu(\theta). \theta_0 \cup \theta \cup \{l' \mid l' \in R\text{-Addr}[\sigma](l) ; l \in \theta\}$ 
  where  $\theta_0 := R_0[\rho](e, \kappa)$ 

 $\text{FV} \in \text{Exp} \rightarrow \text{P}(\text{Var})$ 
 $\text{FV}(x) := \{x\}$ 
 $\text{FV}(i) := \{\}$ 
 $\text{FV}(\lambda(x).e) := \text{FV}(e) - \{x\}$ 
 $\text{FV}(e_1 \text{ op } e_2) := \text{FV}(e_1) \cup \text{FV}(e_2)$ 
 $\text{FV}(\text{if}\theta(e_1)\{e_2\}\{e_3\}) := \text{FV}(e_1) \cup \text{FV}(e_2) \cup \text{FV}(e_3)$ 

 $R_0[\_] \in \text{Env} \rightarrow \text{Exp} \times \text{Kon} \rightarrow \text{P}(\text{Addr})$ 
 $R_0[\rho](e, \kappa) := \{\rho(x) \mid x \in \text{FV}(e)\} \cup R\text{-Kon}[\rho](\kappa)$ 

 $R\text{-Kon}[\_] \in \text{Env} \rightarrow \text{Kon} \rightarrow \text{P}(\text{Addr})$ 
 $R\text{-Kon}[\rho](\kappa) := \{l \mid l \in R\text{-Frame}[\rho](f) ; f \in \kappa\}$ 

 $R\text{-Frame}[\_] \in \text{Env} \rightarrow \text{Frame} \rightarrow \text{P}(\text{Addr})$ 
 $R\text{-Frame}[\rho](\Box \text{ op } e) := \{\rho(x) \mid x \in \text{FV}(e)\}$ 
 $R\text{-Frame}[\rho](v \text{ op } \Box) := R\text{-Val}(v)$ 

 $R\text{-Val} \in \text{Val} \rightarrow \text{P}(\text{Addr})$ 
 $R\text{-Val}(i) := \{\}$ 
 $R\text{-Val}((\lambda(x).e, \rho)) := \{\rho(x) \mid y \in \text{FV}(e) - \{x\}\}$ 

 $R\text{-Addr}[\_] \in \text{Store} \rightarrow \text{Addr} \rightarrow \text{P}(\text{Addr})$ 
 $R\text{-Addr}[\sigma](l) := \{l' \mid l' \in R\text{-Val}(v) ; v \in \sigma(l)\}$ 

 $R[\rho, \sigma](e, \kappa)$  computes the transitively reachable addresses from  $e$  and  $\kappa$  in  $\sigma$ . (We write  $\mu(x). f(x)$  as the least-fixed-point of a function  $f$ .)  $\text{FV}(e)$  computes the free variables for an expression  $e$ .  $R_0[\rho](e, \kappa)$  computes the initial reachable address set for  $e$  and  $\kappa$ .  $R\text{-*}$  computes the reachable address set for a given type.

```

3. Monadic Interpreter

We next design an interpreter for λIF as a monadic interpreter. The monad \mathbb{M} will account for manipulating components of the state space (like Env and Store) and the nondeterminism that arises from abstraction.

```

 $\Sigma := \text{Env} \times \text{Store} \times \text{Kon} \times \text{Time}$ 
 $\mathbb{M}(\alpha) := \Sigma \rightarrow \text{P}(\alpha \times \Sigma)$ 

The monad operation  $\text{bind}$  simultaneously sequence the state  $\Sigma$  and flattens nested nondeterminism. The unit to bind is  $\text{return}$ .

 $\text{bind} : \forall \alpha \beta, \mathbb{M}(\alpha) \rightarrow (\alpha \rightarrow \mathbb{M}(\beta)) \rightarrow \mathbb{M}(\beta)$ 
 $\text{bind}(m)(k)(\zeta) := \{(y, \zeta'') \mid (y, \zeta') \in k(a)(\zeta') ; (a, \zeta') \in m(\zeta)\}$ 

 $\text{return} : \forall \alpha, \alpha \rightarrow \mathbb{M}(\alpha)$ 

```

$\text{return}(a)(\zeta) := \{(a, \zeta)\}$

These operators capture the guts of the explicit state-passing and set comprehension aspects of the interpreter. The rest of the implementation will use these operators and avoid referencing an explicit configuration ζ or sets of results. As is traditional with monadic programming, we use do notation as syntactic sugar for bind . For example:

```
do
  a ← m
  k(a)
```

is just sugar for:

```
bind(m)(k)
```

Interacting with state is achieved through get- and put- effects:

```
get-Env : M(Env)
get-Env((ρ, σ, κ, τ)) := {(ρ, (ρ', σ, κ, τ))}
```

```
put-Env : Env → M(1)
put-Env(ρ')((ρ, σ, κ, τ)) := {(1, (ρ', σ, κ, τ))}
```

(Only get-Env and put-Env are shown for brevity.) Nondeterminism is achieved through null and plus operators $\langle \perp \rangle$ and $\langle + \rangle$:

```
(⊥) : ∀ α, M(α)
(⊥)(ζ) := {}
```

```
_(+)_ : ∀ α, M(α) × M(α) → M α
(m1 (+) m2)(ζ) := m1(ζ) ∪ m2(ζ)
```

The state space for the interpreter is unchanged, although we promote partiality in functions $[\alpha \rightarrow \beta]$ to $[\alpha \rightarrow P(\beta)]$. Values in $P(\beta)$ can be lifted to monadic values $M(\beta)$ using return and $\langle \perp \rangle$, which we name \uparrow_P :

```
↑P : ∀ α, P(α) → M(α)
↑P{a0 .. an} := return(a0) (+) .. (+) return(an)
```

We will also use various coercion helper functions to inject elements of sum types to possibly empty sets:

```
↓cons : Kon → P(Frame × Kon)
↓cons(•) := {}
↓cons(f :: κ) := {f, κ}
```

```
↓clo : Val → P(Clo)
↓clo(i) := {}
↓clo(c) := {c}
```

```
↓Z : Val → P(Z)
↓Z(c) := {}
↓Z(i) := {i}
```

We introduce some helper functions for manipulating the continuation and time components of the state space:

```
push : Frame → M(1)
push(f) := do
  κ ← get-Kon
  put-Kon(f :: κ)
```

```
pop : M(Frame)
pop := do
  κ ← get-Kon
  f, κ' ← ↑P(↓cons(κ))
  put-Kon(κ')
  return(f)
```

```
tick : M(1)
tick = do
  τ ← get-Time
  put-Time(τ + 1)
```

We can now write a monadic interpreter for λIF using these monadic effects.

```
A[ ] ∈ Atom → M(1+Val)
A[i] := return(i)
A[x] := do
  ρ ← get-Env
  σ ← get-Store
  l ← ↑P(ρ(x))
  return(σ(x))
A[λ(x).e] := do
  ρ ← get-Env
  return((λ(x).e, ρ))
```

```
step : Exp → M(Exp)
step(e1 op e2) := do
  tick
  push([□ op e2])
  return(e1)
step(a) := do
  tick
  f ← pop
  v ← A[a]
  case f of
    [□ op e] → do
      push [v op □]
      return(e)
    [v' @ □] → do
      (λ(x).e, ρ') ← ↑P(↓clo(v'))
      l ← alloc(x)
      σ ← get-Store
      put-Env(ρ'[x ↦ l])
      put-Store(σ[l ↦ v])
      return(e)
    [v' iop □] → do
```

```

i1 ←  $\tau_p(\downarrow \mathbb{Z}(v'))$ 
i2 ←  $\tau_p(\downarrow \mathbb{Z}(v))$ 
return( $\delta(iop, i_1, i_2)$ )
[if  $\theta(\square)\{e_1\}\{e_2\}$ ] → do
  i ←  $\tau_p(\downarrow \mathbb{Z}(v))$ 
  if  $i \neq \emptyset$ 
    then return( $e_1$ )
    else return( $e_2$ )

```

To execute a collecting semantics for our interpreter, we form an isomorphism between monadic actions $[\text{Exp} \rightarrow M(\text{Exp})]$ and a the transition system $[P(\Sigma(\text{Exp})) \rightarrow P(\Sigma(\text{Exp}))]$.

```

to : (Exp → M(Exp)) → P(Exp × Σ) → P(Exp × Σ)
to(f)(e $\zeta^*$ ) := {(e,  $\zeta'$ ) | (e,  $\zeta'$ ) ∈ f(e)( $\zeta$ ) ; (e,  $\zeta$ ) ∈ e $\zeta^*$ }

```

```

from : (P(Exp × Σ) → P(Exp × Σ)) → Exp → M(Exp)
from(f)(e)( $\zeta$ ) := f({(e,  $\zeta$ )})

```

Proposition: to and from form an isomorphism.

An collecting semantics is now described as the least-fixed-point of step as transported through the isomorphism:

```

 $\mu(e\zeta^*)$ .  $e\zeta^*_{\emptyset} \cup e\zeta^* \cup \text{to}(\text{step})(e\zeta^*)$ 
where  $e\zeta^*_{\emptyset} := \{(e, \{\downarrow, \downarrow, \bullet, \emptyset\})\}$ 

```

4. A Compositional Framework

In the above monadic interpreter, changes to the language or analysis may require a redesign of the underlying monad. Remarkably, the analysis can be altered to be flow-sensitive by changing the definition of the monad.

```

Σ := Env × Kon × Time
M(α) := Σ × Store → P(α × Σ) × Store

```

```

bind : ∀ α β, M(α) → (α → M(β)) → M(β)
bind(m)(k)( $\zeta, \sigma$ ) := (b $\Sigma^*$ ,  $\sigma''$ )
where
  {(a1,  $\zeta'_1$ ) .. (an,  $\zeta'_n$ )},  $\sigma'$ ) := m( $\zeta, \sigma$ )
  {(b11,  $\zeta''_{11}$ ) .. (bim,  $\zeta''_{im}$ )},  $\sigma''_n$ ) := k(ai)( $\zeta'_i, \sigma'$ )
  b $\Sigma^*$  := {(b11,  $\zeta_{11}$ ) .. (bn1,  $\zeta_{n1}$ ) .. (bnm,  $\zeta_{nm}$ )}
   $\sigma''$  :=  $\sigma''_1 \sqcup \dots \sqcup \sigma''_n$ 

```

```

return : ∀ α, α → M(α)
return(a)( $\zeta, \sigma$ ) := ({a,  $\zeta$ },  $\sigma$ )

```

```

get-Env : M(Env)
get-Env( $\langle \rho, \kappa, \tau \rangle, \sigma$ ) := ({ $\langle \rho, \langle \rho, \kappa, \tau \rangle \rangle$ },  $\sigma$ )

```

```

put-Env : Env → M(1)
put-Env( $\rho'$ )( $\langle \rho, \kappa, \tau \rangle, \sigma$ ) := ({ $\langle 1, \langle \rho', \kappa, \tau \rangle \rangle$ },  $\sigma$ )

```

```

get-Store : M(Env)
get-Store( $\langle \rho, \kappa, \tau \rangle, \sigma$ ) := ({ $\langle \sigma, \langle \rho, \kappa, \tau \rangle \rangle$ },  $\sigma$ )

```

```

put-Store : Store → M(1)
put-Store( $\sigma'$ )( $\langle \rho, \kappa, \tau \rangle, \sigma$ ) := ({ $\langle 1, \langle \rho, \kappa, \tau \rangle \rangle$ },  $\sigma'$ )

```

```

( $\downarrow$ ) : ∀ α, M(α)
( $\downarrow$ )( $\zeta, \sigma$ ) := {}

```

```

_({+})_ : ∀ α, M(α) × M(α) → M α
(m1 {+} m2)( $\zeta, \sigma$ ) := m1( $\zeta, \sigma$ ) ∪ m2( $\zeta, \sigma$ )

```

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 interpreter: state and nondeterminism. There is a monad transformer for adding state effects to existing monads, called the state monad transformer:

```

S $\tau$ [_] : (Type → Type) → (Type → Type)
S $\tau$ [s](m)(α) := s → m (α × s)

```

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

```

bind : ∀ α β, S $\tau$ [s](m)(α) → (α → S $\tau$ [s](m)(β)) → S $\tau$ [s](m)(β)
bind(m)(k)(s) := do
  (x, s') ←m m(s)
  k(x)(s')

```

```

return : ∀ α m, α → S $\tau$ [s](m)(α)
return(x)(s) := returnm(x, s)

```

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

```

get : S $\tau$ [s](m)(s)
get(s) := returnm(s, s)

```

```

put : s → S $\tau$ [s](m)(1)
put(s')(s) := returnm(1, s')

```

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

```

( $\downarrow$ ) : ∀ α, S $\tau$ [s](m)(α)
( $\downarrow$ )(s) := ( $\downarrow$ )m

```

```

_({+})_ : ∀ α, S $\tau$ [s](m)(α) × S $\tau$ [s](m)(α) → S $\tau$ [s](m)(α)
(m1 {+} m2)(s) := m1(s) {+} m2(s)

```

The state monad transformer was introduced by Mark P. Jones in [X]. We have developed a new monad transformer for nondeterminism which can compose with state in both directions.

$P_{\tau} : (\text{Type} \rightarrow \text{Type}) \rightarrow (\text{Type} \rightarrow \text{Type})$
 $P_{\tau}(m)(\alpha) := m(P(\alpha))$

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

$\text{bind} : \forall \alpha \beta, P_{\tau}(m)(\alpha) \rightarrow (\alpha \rightarrow P_{\tau}(m)(\beta)) \rightarrow P_{\tau}(m)(\beta)$
 $\text{bind}(m)(k) := \text{do}$
 $\quad \{x_1 \dots x_n\} \leftarrow m$
 $\quad k(x_1) \sqcup_m \dots \sqcup_m k(x_n)$

$\text{return} : \forall \alpha, \alpha \rightarrow P_{\tau}(m)(\alpha)$
 $\text{return}(x) := \text{return}_m(\{x\})$

Nondeterminism actions $\langle \perp \rangle$ and $\langle + \rangle$ interact with the join-semilattice functoriality of the underlying monad m :

$\langle \perp \rangle : \forall \alpha, P_{\tau}(m)(\alpha)$
 $\langle \perp \rangle := \perp_m$

$_ \langle + \rangle _ : \forall \alpha, P_{\tau}(m)(\alpha) \times P_{\tau}(m)(\alpha) \rightarrow P_{\tau}(m)(\alpha)$
 $m_1 \langle + \rangle m_2 := m_1 \sqcup_m m_2$

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

$\text{get} : P_{\tau}(m)(s)$
 $\text{get} = \text{map}(\lambda(s). \{s\})(\text{get})$

$\text{put} : s \rightarrow P_{\tau}(m)(s)$
 $\text{put}(s) = \text{map}(\lambda(1). \{1\})(\text{put}(s))$

Proposition: P_{τ} 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 monads M have the following operation defined for some Σ :

$\text{mstep} : \forall \alpha \beta, (\alpha \rightarrow M(\beta)) \rightarrow (\Sigma(\alpha) \rightarrow \Sigma(\beta))$

Categorically speaking, `mstep` is a morphism between the Kleisli category for M and the transition system for Σ . We now show that the monad transformers for state and nondeterminism transport this property in addition to monadic operations.

For the state monad transformer $S_{\tau}[s]$ `mstep` is defined:

$\text{mstep} : \forall \alpha \beta m, (\alpha \rightarrow S_{\tau}[s](m)(\beta)) \rightarrow (\Sigma_m(\alpha \times s) \rightarrow \Sigma_m(\beta \times s))$
 $\text{mstep}(f) := \text{mstep}_m(\lambda(a, s). f(a)(s))$

For the nondeterminism transformer P_{τ} `mstep` has two possible definitions. One where Σ is $\Sigma_m \circ P$:

$\text{mstep}_1 : \forall \alpha \beta m, (\alpha \rightarrow P_{\tau}(m)(\beta)) \rightarrow (\Sigma_m(P(\alpha)) \rightarrow \Sigma_m(P(\beta)))$
 $\text{mstep}_1(f) := \text{mstep}_m(\lambda(\{x_1 \dots x_n\}). f(x_1) \langle + \rangle \dots \langle + \rangle f(x_n))$

and one where Σ is $P \circ \Sigma_m$:

$\text{mstep}_2 : \forall \alpha \beta m, (\alpha \rightarrow P_{\tau}(m)(\beta)) \rightarrow (P(\Sigma_m(\alpha)) \rightarrow P(\Sigma_m(\beta)))$
 $\text{mstep}_2(f)(\{\zeta_1 \dots \zeta_n\}) := a\Sigma P_1 \cup \dots \cup a\Sigma P_n$
 where
 $\text{commuteP} : \forall \alpha, \Sigma_m(P(\alpha)) \rightarrow P(\Sigma_m(\alpha))$
 $a\Sigma P_1 := \text{commuteP}(\text{mstep}_m(f)(\zeta_1))$

The operation `commuteP` must be defined for the underlying Σ_m . This property is true for the identity monad, and is preserved by $S_{\tau}[s]$ when Σ_m is also a functor:

$\text{commuteP} : \forall \alpha, \Sigma_m(P(\alpha) \times s) \rightarrow P(\Sigma_m(\alpha \times s))$
 $\text{commuteP} := \text{commuteP}_m \circ \text{map}(\lambda(\{\alpha_1 \dots \alpha_n\}, s). \{(\alpha_1, s) \dots (\alpha_n, s)\})$

We can now build monad transformer stacks from combinations of $S_{\tau}[s]$ and P_{τ} 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 $\Sigma \rightarrow \Sigma$ for some Σ .

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

- $S_{\tau}[\text{Env}] \circ S_{\tau}[\text{Store}] \circ S_{\tau}[\text{Kon}] \circ S_{\tau}[\text{Time}] \circ P_{\tau} \circ \text{ID}$
 - This yields a path-sensitive flow-sensitive analysis.
- $S_{\tau}[\text{Env}] \circ S_{\tau}[\text{Kon}] \circ S_{\tau}[\text{Time}] \circ P_{\tau} \circ S_{\tau}[\text{Store}] \circ \text{ID}$
 - This yields a path-insensitive flow-insensitive analysis coupled with `mstep`.
 - This yields a path-insensitive flow-sensitive analysis coupled with `mstep`.