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 a family of an abstract interpreter. Section X gives the concrete semantics for a small functional language. Section X shows the full definition of a concrete monadic interpreter. Section [X][A Compositional Monadic Framework] shows our compositional metatheory 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}
```

(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. Allocation is made explicit to make the semantics more amenable to abstraction and abstract garbage collection.

The concrete semantics for λIF:

```
τ  ∈ Time   := ℤ

l  ∈ Addr   := Var × Time
ρ  ∈ Env    := Var ⇀ Addr
σ  ∈ Store  := Addr ⇀ Val
c  ∈ Clo    ::= ⟨λ(x).e,ρ⟩
v  ∈ Val    ::= i | c

κl ∈ KAddr  := Time
κσ ∈ KStore := KAddr ⇀ Frame × KAddr
fr ∈ Frame ::= ⟨□ op e⟩ | ⟨v op □⟩ | ⟨if0(□){e}{e}⟩

ς ∈ Σ      ::= Exp × Env × Store × KAddr × KStore

A⟦_,_,_⟧ ∈ Env × Store × Atom ⇀ Val
A⟦ρ,σ,i⟧ := i
A⟦ρ,σ,x⟧ := σ(ρ(x))
A⟦ρ,σ,λ(x).e⟧ := ⟨λ(x).e,ρ⟩

δ⟦_,_,_⟧ ∈ IOp × ℤ × ℤ → ℤ
δ⟦+,i₁,i₂⟧ := i₁ + i₂
δ⟦-,i₁,i₂⟧ := i₁ - i₂

_-->_ ∈ P(Σ × Σ)
⟨e₁ op e₂,ρ,σ,κl,κσ,τ⟩ --> ⟨e₁,ρ,σ,τ,κσ',tick(τ)⟩
  where κσ' := κσ[τ ↦ ⟨□ op e₂⟩∷κl]
⟨a,ρ,σ,κl,κσ,τ⟩ --> ⟨e,ρ,σ,τ,κσ',tick(τ)⟩
  where ⟨□ op e⟩∷κl' := κσ(κl)
        κσ' := κσ[τ ↦ ⟨A⟦ρ,σ,a⟧ op □⟩∷κl']
⟨a,ρ,σ,κl,κσ,τ⟩ --> ⟨e,ρ'',σ',κl',κσ,tick(τ)⟩
  where ⟨⟨λ(x).e,ρ'⟩ @ □⟩∷κl':= κσ(κl)
        σ' := σ[(x,τ) ↦ A⟦ρ,σ,a⟧]
        ρ'' := ρ'[x ↦ (x,τ)]
⟨i₂,ρ,σ,κl,κσ,τ⟩ --> ⟨i,ρ,σ,κl',κσ,tick(τ)⟩
  where ⟨i₁ iop □⟩∷κl' := κσ(κl)
        i := δ⟦iop,i₁,i₂⟧
⟨i,ρ,σ,κl,κσ,τ⟩ --> ⟨e,ρ,σ,κl',κσ,tick(τ)⟩
  where ⟨if0(□){e₁}{e₂}⟩∷κl' := κσ(κl)
        e := if(i = 0) then e₁ else e₂
```

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

```
_~~>_ ∈ P(Σ × Σ)
ς ~~> ς'
  where ς --> ς'
⟨e,ρ,σ,κl,κσ,τ⟩ ~~> ⟨e,ρ,σ',κl,κσ,τ⟩
  where σ' := {l ↦ σ(l) | l ∈ R[σ](ρ,e)}
        κσ' := {κl ↦ κσ(κl) | κl ∈ κR[κσ](κl)}
```

where R is the set of addresses reachable from a given expression:

```
R[_] ∈ Store → Env × Exp → P(Addr)
R[σ](ρ,e) := μ(θ). R₀(ρ,e) ∪ θ ∪ {l' | l' ∈ R-Val(σ(l)) ; l ∈ θ}

R₀ ∈ Env × Exp → P(Addr)
R₀(ρ,e) := {ρ(x) | x ∈ FV(e)}

FV ∈ Exp → P(Var)
FV(x) := {x}
FV(i) := {}
FV(λ(x).e) := FV(e) - {x}
FV(e₁ op e₂) := FV(e₁) ∪ FV(e₂)
FV(if0(e₁){e₂}{e₃}) := FV(e₁) ∪ FV(e₂) ∪ FV(e₃)

R-Val ∈ Val → P(Addr)
R-Val(i) := {}
R-Val(⟨λ(x).e,ρ⟩) := {ρ(x) | y ∈ FV(λ(x).e)}
```

R[σ](ρ,e) computes the transitively reachable addresses from e in ρ and σ. (We write μ(x). f(x) as the least-fixed-point of a function f.) R₀(ρ,e) computes the initial reachable address set for e under ρ. FV(e) computes the free variables for an expression e. R-Val computes the addresses reachable from a value.

Analagously, κR is the set of addresses reachable from a given continuation address:

```
κR[_] ∈ KStore → KAddr → P(KAddr)
κR[κσ](κl) := μ(kθ). κθ₀ ∪ κθ ∪ { π₂(κR-Frame(κσ(κl))) | κl ∈ κθ}
```

3. Monadic Interpreter

We next design an interpreter for λIF as a monadic interpreter. This interpreter will support both concrete and abstract executions. To do this, there will be three parameters which the user can instantiate in any way they wish:

1. The monad, which captures the flow-sensitivity of the analysis.
2. The value space, which captures the abstract domain for integers and closures.
3. Time and address, which captures the call-site sensitivity of the analysis.

We place each of these features behind an abstract interface. For now we leave the implementations of these interfaces opaque. We will recover specific concrete and abstract interpreters in a later section.

The goal is to implement as much of the interpreter as possible while leaving these things abstract. The more we can prove about our interpreter independent of these variables, the more proof-work we'll get for free.

3.1 The Monad Interface

The interpreter will use a monad `M` in two ways. First, to manipulate components of the state space (like `Env` and 'Store). Second, to exhibit nondeterministic behavior, which is inherent in computable analysis. We capture these properties as monadic effects.

To be a monad, `M` must have type:

```
M : Type → Type
```

and support the `bind` operation:

```
bind : ∀ α β, M(α) → (α → M(β)) → M(β)
```

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

```
return : ∀ α, α → M(α)
```

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

```
bind-unit₁ : bind(return(a))(k) = k(a)
bind-unit₂ : bind(aM)(return) = aM
bind-associativity : bind(bind(aM)(k₁))(k₂) = bind(aM)(λ(a)→bind(k₁(a))(k₂))
```

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

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

```
do
  a ← m
  k(a)
```

and

```
a ← m ; k(a)
```

are both just sugar for

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

Interacting with `Env` is achieved through `get-Env` and `put-Env` effects:

```
get-Env : M(Env)
put-Env : Env → M(1)
```

which have the following laws:

```
put-put : put-Env(s₁) ; put-Env(s₂) = put-Env(s₂)
put-get : put-Env(s) ; get-Env = return(s)
get-put : s ← get-Env ; put-Env(s) = return(1)
get-get : s₁ ← get-Env ; s₂ ← get-Env ; k(s₁,s₂) = s ← get-Env ; k(s,s)
```

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

Nondeterminism is achieved through operators `⟨⊥⟩` and `⟨+⟩`:

```
⟨⊥⟩ : ∀ α, M(α)
_⟨+⟩_ : ∀ α, M(α) × M(α) → M α
```

which have the following laws:

```
⊥-zero₁ : bind(⟨⊥⟩)(k) = ⟨⊥⟩
⊥-zero₂ : bind(aM)(λ(a)→⟨⊥⟩) = ⟨⊥⟩
⊥-unit₁ : ⟨⊥⟩ ⟨+⟩ aM = aM
⊥-unit₂ : aM ⟨+⟩ ⟨⊥⟩ = aM
+-associativity : aM₁ ⟨+⟩ (aM₂ ⟨+⟩ aM) = (aM₁ ⟨+⟩ aM₂) ⟨+⟩ aM
+-commutativity : aM₁ ⟨+⟩ aM₂ = aM₂ ⟨+⟩ aM₁
+-distributivity : bind(aM₁ ⟨+⟩ aM₂)(k) = bind(aM₁)(k) ⟨+⟩ bind(aM₂)(k)
```

3.2 The Value Space Interface

To abstract the value space we require the type `Val` be an opaque parameter We need only require that `Val` is a join-semilattice:

```
⊥ : Val
_▯_ : Val × Val → Val
```

The interface for integers consists of introduction and elimiation rules:

```
int-I : ℤ → Val
int-if0-E : Val → P(▯)
```

We can now state laws for this interface, which are designed to induce a Galois connection between `ℤ` and `Val`:

```
{true}  ⊑ int-if0-E(int-I(i))     if i = 0
{false} ⊑ int-if0-E(int-I(i))     if i ≠ 0


v ⊒ ▯_{b ∈ int-if0-E(v)} θ(b)
  where θ(true)  = int-I(0)
        θ(false) = ▯_{i ∈ ℤ | i ≠ 0} int-I(i)
```

Additionally we must abstract closures:

```
clo-I : Clo → Val
clo-E : Val → P(Clo)
```

which follow similar laws:

```
{c} ⊑ clo-E(cloI(c))
v ⊑ □_{c ∈ clo-E(v)} clo-I(c)
```

The denotation for primitive operations `δ` must also be opaque:

```
δ : IOp × Val × Val → Val
```

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.

3.3 Abstract Time and Addresses

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

```
alloc : Var × Time → Addr
κalloc : Time → KAddr
tick : Exp × Kon × Time → 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 `Time` and `Addr` to support 0-CFA and k-CFA, so we specialize `Σ` to `Exp × Kon`.

Remarkably, we need not state laws for `alloc` and `tick`. Our interpreter will always merge values which reside at the same address to achieve soundness. Therefore, any supplied implementations of `alloc` and `tick` are valid.

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`:

```
σ  ∈ Store  : Addr → Val
κσ ∈ KStore : KAddr → P(Frame × KAddr)
```

We have already established a join-semilattice structure for `Val`. Developing a custom join-semilattice for continuations is possible. Fro this presentation we use `P(Frame × KAddr)` for simplicity.

3.4 Interpreter Definition

We use the three interfaces from above as opaque parameters to out interpreter. Before defining the interpreter we define three helper functions. These helper functions crucially rely on the monadic effect interface.

First, values in `P(β)` can be lifted to monadic values `M(β)` using `return` and `⟨⊥⟩`, which we name `↑ₚ`:

```
↑ₚ : ∀ α, P(α) → M(α)
↑ₚ({a₁ .. aₙ}) := return(a₁) ⟨+⟩ .. ⟨+⟩ return(aₙ)
```

We provide an elimination rule from lists of frames into a singleton set of a cons cell:

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

We introduce monadic helper functions for allocation and manipulating time:

```
allocM : Var → M(Addr)
allocM(x) := do
  τ ← get-Time
  return(alloc(x,τ))

κallocM : M(KAddr)
κallocM := do
  τ ← get-Time
  return(κalloc(τ))

tickM : Exp → M(1)
tickM(e) = do
  τ ← get-Time
  κ ← get-Kon
  put-Time(tick(e,κ,τ))
```

Finally we introduce helper functions for manipulating stack frames:

```
push : Frame → M(1)
push(fr) := do
  κl ← get-KAddr
  κσ ← get-KStore
  κl' ← κallocM
  put-KStore(κσ □ [κl' ↦ {fr∷κl}])
  put-KAddr(κl')

pop : M(Frame)
pop := do
  κl ← get-KAddr
  κσ ← get-KStore
  fr∷κl' ← ↑ₚ(κσ(κl))
  put-KAddr(κl')
  return(fr)
```

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

```
A⟦_⟧ ∈ Atom → M(Val)
A⟦i⟧ := return(int-I(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(clo-I(⟨λ(x).e,ρ⟩))

step : Exp → M(Exp)
step(e₁ op e₂) := do
  tickM(e₁ op e₂)
  push([□ op e₂])
  return(e₁)
step(a) := do
  tickM(a)
  f ← pop
  v ← A⟦a⟧
  case f of
    [□ op e] → do
      push([v op □])
      return(e)
    [v' @ □] → do
      ⟨λ(x).e,ρ'⟩ ← ↑ₚ(clo-E(v'))
      l ← alloc(x)
      σ ← get-Store
      put-Env(ρ'[x↦l])
      put-Store(σ[l↦v])
      return(e)
    [v' iop □] → do
      return(δ(iop,v',v))
    [if0(□){e₁}{e₂}] → do
      b ← ↑ₚ(int-if0-E(v))
      if(b) then return(e₁) else return(e₂)
```

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 `(Σ → Σ)α⇄γ(Exp → M(Exp))`. This Galois connection serves two purposes. First, it allows us to implement the analysis by converting our interpreter to the transition system `Σ → Σ` through `γ`. Second, this Galois connection serves to *transport other Galois connections*. For example, when supplied with concrete and abstract versions of □□□, we carry □□□ α⇄γ ^□□□^ through the galois connection to establish □ α⇄γ ^□^. We go into more detail on this point in Section [X][The Proof Framework].

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

```
μ(ς). ς₀ □ ς □ γ(step)(ς)
```

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

4. Recovering Anlayses

We now instantiate our monadic interpreter from Section X. We will recover a conrete interpreter and a family of abstract interpreter. Furthermore, we will prove each of the abstract interpreter correct with strikingly minimal proof burden.

4.1 Recovering a Concrete Interpreter [UNDER CONSTRUCTION]

To recover a concrete interpreter we instantiate `M` to a powerset of products state monad:

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

bind(m)(k)(ς) := {(y,ς'') | (y,ς'') ∈ k(a)(ς') ; (a,ς') ∈ m(ς)}
return(a)(ς) := {(a,ς)}

get-Env(⟨ρ,σ,κ,τ⟩) := {(ρ,⟨ρ,σ,κ,τ⟩)}
put-Env(ρ')(⟨ρ,σ,κ,τ⟩) := {(1,⟨ρ',σ,κ,τ⟩)}

⟨⊥⟩(ς) := {}
(m₁ ⟨+⟩ m₂)(ς) := m₁(ς) ∪ m₂(ς)

to : (Exp → M(Exp)) → P(Exp × Σ) → P(Exp × Σ)
to(f)(eς*) := {(e,ς') | (e,ς') ∈ f(e)(ς) ; (e,ς) ∈ eς*}

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

  where eς*₀ := {(e,⟨⊥,⊥,•,0⟩)}
```

5. 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)(ς,σ) := (bΣ*,σ''')
  where
    ({(a₁,ς'₁) .. (aₙ,ς'ₙ)},σ') := m(ς,σ)
    ({(bᵢ₁,ς''ᵢ₁) .. (bᵢₘ,ς''ᵢₘ)},σ''ₙ) := k(aᵢ)(ς'ᵢ,σ')
    bΣ* := {(b₁₁,ς₁₁) .. (bₙ₁,ςₙ₁) .. (bₙₘ,ςₙₘ)}
    σ''' :=  σ''₁ □ .. □ σ''ₙ

return : ∀ α, α → M(α)
return(a)(ς,σ) := ({a,ς},σ)

get-Env : M(Env)
get-Env(⟨ρ,κ,τ⟩,σ) := ({(ρ,⟨ρ,κ,τ⟩)},σ)

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

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

put-Store : Store → M(1)
put-Store(σ')(⟨ρ,κ,τ⟩,σ) := ({(1,⟨ρ,κ,τ⟩)},σ')

⟨⊥⟩ : ∀ α, M(α)
⟨⊥⟩(ς,σ) := {}

_⟨+⟩_ : ∀ α, M(α) × M(α) → M α
(m₁ ⟨+⟩ m₂)(ς,σ) := m₁(ς,σ) ∪ m₂(ς,σ)
```

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:

```
Sₜ[_] : (Type → Type) → (Type → Type)
Sₜ[s](m)(α) := s → m (α × s)
```

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

```
bind : ∀ α β, Sₜ[s](m)(α) → (α → Sₜ[s](m)(β)) → Sₜ[s](m)(β)
bind(m)(k)(s) := do
  (x,s') ←ₘ m(s)
  k(x)(s')

return : ∀ α m, α → Sₜ[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 : Sₜ[s](m)(s)
get(s) := returnₘ(s,s)

put : s → Sₜ[s](m)(1)
put(s')(s) := returnₘ(1,s')
```

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

```
⟨⊥⟩ : ∀ α, Sₜ[s](m)(α)
⟨⊥⟩(s) := ⟨⊥⟩ₘ

_⟨+⟩_ : ∀ α, Sₜ[s](m)(α) x Sₜ[s](m)(α) → Sₜ[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.

```
Pₜ : (Type → Type) → (Type → Type)
Pₜ(m)(α) := m(P(α))
```

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

```
bind : ∀ α β, Pₜ(m)(α) → (α → Pₜ(m)(β)) → Pₜ(m)(β)
bind(m)(k) := do
  {x₁ .. xₙ} ←ₘ m
  k(x₁) □ₘ .. □ₘ k(xₙ)

return : ∀ α, α → Pₜ(m)(α)
return(x) := returnₘ({x})
```

Nondterminism actions `⟨⊥⟩` and `⟨+⟩` interact with the join-semilattice functorality of the underlying monad `m`:

```
⟨⊥⟩ : ∀ α, Pₜ(m)(α)
⟨⊥⟩ := ⊥ₘ

_⟨+⟩_ : ∀ α, Pₜ(m)(α) x Pₜ(m)(α) → Pₜ(m)(α)
m₁ ⟨+⟩ m₂ := m₁ □ₘ m₂
```

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

```
get : Pₜ(m)(s)
get = map(λ(s).{s})(get)

put : s → Pₜ(m)(s)
put(s) = map(λ(1).{1})(put(s))
```

Proposition: `Pₜ` 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`:

```
mstep : ∀ α β, (α → M(β)) → (Σ(α) → Σ(β))
```

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 `Sₜ[s]` mstep is defined:

```
mstep : ∀ α β m, (α → Sₜ[s](m)(β)) → (Σₘ(α × s) → Σₘ(β × s))
mstep(f) := mstepₘ (λ(a,s). f(a)(s))
```

For the nondeterminism transformer `Pₜ` mstep has two possible definitions. One where `Σ` is `Σₘ ∘ P`:

```
mstep₁ : ∀ α β m, (α → Pₜ(m)(β)) → (Σₘ(P(α)) → Σₘ(P(β)))
mstep₁(f) := mstepₘ(λ({x₁ .. xₙ}). f(x₁) ⟨+⟩ .. ⟨+⟩ f(xₙ))
```

and one where `Σ` is `P ∘ Σₘ`:

```
mstep₂ : ∀ α β m, (α → Pₜ(m)(β)) → (P(Σₘ(α)) → P(Σₘ(β)))
mstep₂(f)({ς₁ .. ςₙ}) := aΣP₁ ∪ .. ∪ aΣPₙ
  where
    commuteP : ∀ α, Σₘ(P(α)) → P(Σₘ(α))
    aΣPᵢ := commuteP(mstepₘ(f)(ςᵢ))
```

The operation `computeP` must be defined for the underlying `Σₘ`. This property is true for the identiy monad, and is preserved by `Sₜ[s]` when `Σₘ` is also a functor:

```
commuteP : ∀ α, Σₘ(P(α) × s) → P(Σₘ(α × s))
commuteP := commutePₘ ∘ map(λ({α₁ .. αₙ},s). {(α₁,s) .. (αₙ,s)})
```

We can now build monad transformer stacks from combinations of `Sₜ[s]` and `Pₜ` 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 `Σ → Σ` for some `Σ`.

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

- `Sₜ[Env] ∘ Sₜ[Store] ∘ Sₜ[Kon] ∘ Sₜ[Time] ∘ Pₜ ∘ ID`
  - This yields a path-sensitive flow-sensitive analysis.
- `Sₜ[Env] ∘ Sₜ[Kon] ∘ Sₜ[Time] ∘ Pₜ ∘ Sₜ[Store] ∘ ID`
  - This yields a path-insensitive flow-insensitive analysis coupled with mstep .
  - This yeilds a path-insensitive flow-sensitive analysis coupled with mstep .