Modular Metatheory for Abstract Interpreters

Abstract -

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.
- 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}
```

(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`:

```
τ ∈ Time   := ℤ
l ∈ Addr   := Var × ℤ
ρ ∈ Env    := Var ⇀ Addr
σ ∈ Store  := Addr ⇀ Val
f ∈ Frame ::= [□ op e] | [v op □] | [if0(□){e}{e}]
κ ∈ Kon    := Frame*
c ∈ Clo   ::= ⟨λ(x).e,ρ⟩
v ∈ Val   ::= i | c
ς ∈ Σ     ::= Exp × Env × Store × Kon

alloc ∈ Var × Time → Addr
alloc(x,τ) := ⟨x,τ⟩

tick ∈ Time → Time
tick(τ) := τ + 1

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,κ,τ⟩ --> ⟨e₁,ρ,σ,[□ op e₂]∷κ,tick(τ)⟩
⟨a,ρ,σ,[□ op e]∷κ,τ⟩ --> ⟨e,ρ,σ,[v op □]∷κ,tick(τ)⟩
  where v = A⟦ρ,σ,a⟧
⟨a,ρ,σ,[v₁ @ □]∷κ,τ⟩ --> ⟨e,ρ'[x↦l],σ[l↦v₂],κ,tick(τ)⟩
  where ⟨λ(x).e,ρ'⟩ := v₁
        v₂ := A⟦ρ,σ,a⟧
        l := alloc(x,τ)
⟨i₂,ρ,σ,[i₁ iop □]∷κ,τ⟩ --> ⟨i,ρ,σ,κ,tick(τ)⟩
  where i := δ⟦iop,i₁,i₂⟧
⟨i,ρ,σ,[if0(□){e₁}{e₂}]∷κ,τ⟩ --> ⟨e,ρ,σ,κ,tick(τ)⟩
  where e := e₁ if i = 0
             e₂ otherwise
```

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,κ,τ⟩ ~~> ⟨e,ρ,{l↦σ(l) | l ∈ R[ρ,σ](e,κ)},κ,τ⟩

R[_,_] ∈ Env × Store → Exp × Kon → P(Addr)
R[ρ,σ](e,κ) := μ(θ). θ₀ ∪ θ ∪ {l' | l' ∈ R-Addr[σ](l) ; l ∈ θ}
  where θ₀ := R₀[ρ](e,κ)

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)

R₀[_] ∈ Env → Exp × Kon → P(Addr)
R₀[ρ](e,κ) := {ρ(x) | x ∈ FV(e)} ∪ R-Kon[ρ](κ)

R-Kon[_] ∈ Env → Kon → P(Addr)
R-Kon[ρ](κ) := {l | l ∈ R-Frame[ρ](f) ; f ∈ κ}

R-Frame[_] ∈ Env → Frame → P(Addr)
R-Frame[ρ](□ op e) := {ρ(x) | x ∈ FV(e)}
R-Frame[ρ](v op □) := R-Val(v)

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

R-Addr[_] ∈ Store → Addr → P(Addr)
R-Addr[σ](l) := {l' | l' ∈ R-Val(v) ; v ∈ σ(l)}
```

`R[ρ,σ](e,κ)` computes the transitively reachable addresses from `e` and `κ` in `σ`. (We write `μ(x). f(x)` as the least-fixed-point of a function `f`.) `FV(e)` computes the free variables for an expression `e`. `R₀[ρ](e,κ)` computes the initial reachable address set for `e` and `κ`. `R-*` 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 `M` will account for manipulating components of the state space (like `Env` and `Store`) and the nondeterminism that arises from abstraction.

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

The monad operation `bind` simultaneously sequence the state `Σ` and flattens nested nondeterminism. The unit to `bind` is `return`.

```
bind : ∀ α β, M(α) → (α → M(β)) → M(β)
bind(m)(k)(ς) := {(y,ς'') | (y,ς'') ∈ k(a)(ς') ; (a,ς') ∈ m(ς)}

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

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 `ς` 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 `⟨⊥⟩` and `⟨+⟩`:

```
⟨⊥⟩ : ∀ α, M(α)
⟨⊥⟩(ς) := {}

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

The state space for the interpeter is unchanged, although we promote partiality in functions `[α ⇀ β]` to `[α → P(β)]`. 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({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 compoments 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 ← ↑ₚ(↓ℤ(v'))
      i₂ ← ↑ₚ(↓ℤ(v))
      return(δ(iop,i₁,i₂))
    [if0(□){e₁}{e₂}] → do
      i ← ↑ₚ(↓ℤ(v))
      if i ≟ 0
        then return(e₁)
        else return(e₂)
```

To execute a collecting semantics for our interpreter, we form an isomorphism between monadic actions `[Exp → M(Exp)]` and a the transition system `[P(Σ(Exp)) → P(Σ(Exp))]`.

```
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)})
```

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:

```
μ(eς*). eς*₀ ∪ eς* ∪ to(step)(eς*)
  where eς*₀ := {(e,⟨⊥,⊥,•,0⟩)}
```

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)(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 .