Modular Metatheory for Abstract Interpreters

1. Introduction

Writing abstract interpreters is hard. Establishing the proof of correctness of an abstract interpreter is even harder. Modern practice in whole-program analysis requires multiple iterations of abstract models during the design process. What we lack is a meta-theory framework for designing new abstract interpreters that come with correctness proofs.

We propose a compositional meta-theory framework for static analysis. Our framework gives the analysis designer building blocks for building static analysis. These building blocks are highly compositional, and carry both computational and correctness properties of an analysis. Analyses built in our framework enjoy two key properties not present in previous work:

- Analyses are correct-by-construction.
- The path and flow sensitivities of an analysis can be recovered through plug-and-play modules.

Our framework leverages monad transformers as the fundamental building blocks for an abstract interpreter. Monad transformers compose to form a single monad which underlies a monadic abstract interpreter. Each piece of the monad transformer stack corresponds to an element of the semantics' state space. Variations in the stack are shown to give rise to different path and flow sensitivities.

The *monad* abstraction provides both computational and proof properties for interpreters. The operations provide an abstraction for computation, and the monad laws provide a framework for proof. The *monad transformer* are actions which build monads piece-wise. Monad transformers are just compositional monads. We prove that any instantiation of monad transformers in our framework results in a correct-by-construction abstract interpreter.

1.1 Contributions:

Our contributions are:

- A compositional meta-theory framework for building correct-by-construction abstract interpreters.
- A new monad transformer for nondeterminism.
- An isolated understanding of flow-sensitivity as variations in the monad underlying an interpreter.

1.2 Outline

We demonstrate our framework by example: we walk the reader through the design and implementation of a family of correct-by-construction abstract interpreters. Section 2 gives the concrete semantics for a small functional language. Section 3 sketches the correct-by-construction methodology of our framework Section 4 shows the concrete monadic interpreter. Section 5 performs systematic abstraction of the interpreter to enable a wide range of analyses.

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. Methodology

To design abstract interpreters for `λIF` we adhere to the following methodology:

1. Parameterize over some element of the state space (`Val`, `Addr`, `M`, etc.) and its operations.
   - Show that the interpreter is monotonic w.r.t. the parameters.
     - *i.e.*, if `[Val α⇄γ ^Val^]` and `[+ ⊑ γ ∘ ^+^ ∘ α]` then `[step(Val) α⇄γ step(^Val^)]`.
2. Relate the interpreter to a state space transition system.
   - Show that the mapping between the interpreter and transition system preserves Galois connections.
   - Show that the abstract state space is finite, and therefore that the analysis is computable.
   - An analysis is the least-fixed-point solution to the (finite) transition system.
3. Recover the concrete semantics and design a family of abstractions.
   - Show that there are choices which have Galois connections.

    - *i.e.*, `[Val α⇄γ ^Val^]`.
  - Show that abstract operators are approximations of concrete ones.
    - *i.e.*, `[+ ⊑ γ ∘ ^+^ ∘ α]`.

Following the above methodology results in end-to-end correctness proofs for abstract interpreters. We show how to obtain items 1 and 2 for free using compositional building blocks. Our building blocks snap together to construct both computational and correctness components of an analysis.

First we will introduce our compositional building blocks for building correct-by-construction abstract interpreters. Then we will apply item 3 to three orthogonal design axis:

- The monad `M` for the interpreter, exposing the *flow sensitivity* of the analysis. Exposing this axis is novel to this work.
- The abstract value space `Val` for the interpreter, exposing the *abstract domain* of the analysis.
- The choice for `Time` and `Addr`, exposing the *call-site sensitivity* of the analysis.

The rest of the paper is as follows:

1. We begin by writing a monadic concrete interpreter for `λIF`.
   - There are no parameters to the interpreter yet.
   - We show how to relate the monadic concrete interpreter to an executable state space transition system.
2. We then introduce our compositional framework for building abstract interpreters.
   - Our framework leverages monad transformers as vehicles for transporting both computation and proofs of correctness.
   - We apply the framework to `λIF`, although the tools are directly usable for other languages and analyses.
3. We parameterize over `M` and monadic effects `get`, `put`, `⊥` and `⟨+⟩` in the interpreter, exposing *flow sensitivity*.
   - We show that our interpreter is monotonic w.r.t. `M` and monadic effects.
   - We instantiate `M` with `path-sensitive ⊑ flow-sensitive ⊑ flow-sensitive` implementations.
4. We parameterize over `Val` and `δ` in the interpreter, exposing the *abstract domain*.
   - We show that the interpreter is monotonic w.r.t. `Val` and `δ`.
   - We instantiate `ℤ` in Val with `ℤ ⊑ {-,0,+}`.
5. We parameterize over `Time`, `Addr`, `alloc` and `tick` in the interpreter, exposing *call-site sensitivity*.
   - We show that the interpreter is monotonic w.r.t. `Addr`, `Time` and their operations.
   - We instantiate `[Time × Addr]` with `[Exp* × (Var × Exp*)] ⊑ [Exp*ₖ × (Var × Exp*ₖ)] ⊑ [1 × (Var × 1)]`.
6. We observe that the implementation *and proof of correctness* for abstract garbage require no change as we vary each parameter.

4. Monadic Interpreter

We design an interpreter for `λIF` as a monadic interpreter. The monadic abstract will be used to encode both state effects and nondeterminism.

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

The basic monad operations `return` and `bind` sequence the state `Σ` and provide generic plumbing.

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

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

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

```
do
  x ← m
  k(x)
```

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 encode partiality in functions `[α ⇀ β]` explicitly as `[α → 1+β]`. Pointed values can be lifted to monadic values using `return` and `⟨⊥⟩`, which we name $\uparrow_1$:

```
↑₁ : ∀ α, 1+α → M(α)
↑₁(inl(1)) := ⟨⊥⟩
↑₁(inr(x)) := return(x)
```

We will also use various coercion helper functions to inject elements of sum types to a pointed branch:

```
↓cons : Kon → 1+Frame × Kon
↓cons(•) := inl(1)
↓cons(f∷κ) := inr(f,κ)

↓clo : Val → 1+Clo
↓clo(i) := inl(1)
↓clo(c) := inr(c)

↓ℤ : Val → 1+ℤ
↓ℤ(c) := inl(1)
↓ℤ(i) := inr(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' ← ↑₁(↓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 ← ↑₁(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'⟩ ← ↑₁(↓clo(v'))
      l ← alloc(x)
      σ ← get-Store
      put-Env(ρ'[x↦l])
      put-Store(σ[l↦v])
      return(e)
    [v' iop □] → do
      i₁ ← ↑₁(↓ℤ(v'))
      i₂ ← ↑₁(↓ℤ(v))
      return(δ(iop,i₁,i₂))
    [if0(□){e₁}{e₂}] → do
      i ← ↑₁(↓ℤ(v))
      if i ≟ 0
        then return(e₁)
        else return(e₂)
```

To execute our analysis, 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 analysis is now described as the least-fixed-point of a collecting semantics of `step` as transported through the isomorphism:

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

Thi isomorphism between monadic actions and the transition system will be key to our compositional proof framework.

4.1 Monad Transformers: Compositional Building Blocks

5. Systematic Abstraction