Modular Metatheory for Abstract Interpreters

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*
v ∈ Val   ::= i | ⟨λ(x).e,ρ⟩
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).
  R₀[ρ](e,κ) ∪ θ ∪ {l' | l' ∈ R-Addr[σ](l) | l ∈ θ}

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.

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` and 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.

Contributions:

- A compositional framework for building both computations and proofs for abstract interpreters.
  - We leverage monad transformers to transport both computation and proof.
  - A new monad transformer for nondeterminism.
- Carving out flow-sensitivity as orthogonal to other analysis features like abstract domain and call-site sensitivity.
  - Variations in flow-sensitivity are understood in isolation as mere variations in the monad used for the interpreter.