A Modular Monadic Framework for Abstract Interpretation ("Guts" Draft)

David Darais

April 3, 2014

Abstract

Abstract interpretation is important. *Modular* frameworks for *deriving* computable abstract interpreters are also important. Previous work[PLDI 2013] demonstrates a framework that unifies several concepts in abstract interpretation through monads. However this approach lacks true modularity, both in the space of languages and the space of analyses, and the correctness of the approach is established (if at all) in an ad-hoc basis. This paper demonstrated a reformulation of the monadic frameork for abstract interpretation which is truly modular and where derived abstract interpreters are correct by construction. The key players responsible for recovering these qualities are a restricted class of monad transformers which are also Galois functor transformers, and whose monadic actions are isomorphic to a space of transitions in a pure (non-monadic) state space.

1 Introduction

2 Overview of Abstract Interpretation

3 Overview of Monadic Abstract Interpretation

4 A new approach to Monadic Abstract Interpretation

Our approach slices the monads of the existing approach into reusable, language-agnostic monad transformer stacks. In Section ? we will establish exactly the space of monad transformers which, given specific properties can be proven about them, will integrate seamlessly within our framework. For now we introduce a few familiar monad transformers for the sake of demonstration, and defer the explanation of *why* they can be used.

```
  Identity       A := A
  StateT    S m A := S → m (S × A)
  State     S   A := StateT Identity A
  Set           A := {... , A, ...}
  SetT        m A := m (Set A)
```

Next, we introduce interfaces for various monadic behaviors and their laws:

```
Monad (m : Type -> Type) :=
  return : ∀ A, A → m A
  bind : ∀ A B, m A → (A → m B) → m B
  left_unit : bind aM ret = aM
  right_unit : bind (ret x) f = f x
  associativity : bind (bind aM f) g = bind aM (λ x → bind (f x) g)
MonadFunctor (t : (Type -> Type) -> Type -> Type):=
  mapMonad : ∀ m, Monad m -> Monad (t m)
MonadPlus m | Monad m :=
  mzero : ∀ A, m A
  _<+>_ : ∀ A, m A -> m A -> m A
  unit : aM <+> mzero = aM
  left_zero : bind mzero f = mzero
  right_zero : bind aM (λ x → mzero) = mzero
  associativity : (aM <+> bM) <+> cM = aM <+> (bM <+> cM)
  commutativity : aM <+> bM = bM + aM
MonadPlusFunctor t | MonadFunctor t :=
  mapMonadPlus : ∀ S m, MonadPlus m -> MonadPlus (t m)
MonadState (S : Type) m | Monad m :=
  getState : m S
  putState : S → m unit
MonadStateFunctor t | MonadFunctor t :=
  mapMonadState : ∀ S m, MonadState S m -> MonadState S (t m)
```

and prove the following lemmas:

```
forall S:
  - MonadFunctor (StateT S)
  - MonadPlusFunctor (StateT S)
  - MonadStateFunctor (StateT S)
forall S and m | Monad m:
  - MonadState S (StateT S m)

MonadFunctor SetT
MonadStateFunctor SetT
∀ m | Monad m:
  - MonadPlus (SetT m)
```

We define **mjoin** for transporting values from the Set monad to an arbitrary MonadPlus:

```
mjoin : forall A m | MonadPlus m, Set A -> m A
mjoin {} = mzero
mjoin {..., x, ...} = ... <+> return x <+> ...
```

4.1 Methodology

Languages are embedded in the framework using the following methodology:

- Define a *single* language state space, to be used for both concrete and abstract interpreters.
- Define a *single monadic* small-step function, parameterized by a monadic effect interface and δ function.

For example, consider the following state space for a very simple arithmetic language:

```
Int := { ... the integers ... }
Symbol := { ... x, y, etc ... }
Sign := IsZero | IsPos | IsNeg
AInt := Exact Int | Symbolic Sign
Lang := Halt Atomic
      | LetInc Symbol Atomic Lang
      | IfZero Atomic Lang Lang
Atomic := Num AInt | Var Symbol
Env A B := A -> Option B
```

Notice that **Lang** is a *mixed* concrete-abstract language, potentially capable of expressing both concrete and abstract semantics. In this example we will call the delta function **inc**, because it interprets increment for atomic expressions. Given ⟨**m, inc**⟩ where:

```
– Monad m
– MonadPlus m
– HasState (Env Symbol (Set AInt)) m
– inc : AInt → AInt → Set Ant
```

The semantics for the language **AInt** can be defined:

```
lookup :: Symbol -> Env Symbol (dom AInt) -> m (dom AInt)
lookup x e = case e x of
  None -> mzero
  Some aD -> return aD

atomic : Atomic -> m (dom AInt)
atomic (Num n) = return (return n)
atomic (Var x) = do
  e <- getEnv
  v <- lookup x e
  return v

step : Lang -> m Lang
step (Halt a) = return (Halt a)
step (LetInc x na l) = do
  n <- mjoin (atomic na)
  r <- delta n
  modifyEnv (insert x r)
  return l
step (IfZero ca tb fb) = do
  c <- mjoin (atomic ca)
  case c of
    Exact 0 -> return tb
    Exact _ -> return fb
    Symbolic IsZero -> return tb
    Symbolic IsNeg -> return fb
    Symbolic IsPos -> return fb
```

This single semantic step function is capable of expressing both the concrete and abstract semantics of our arithmetic language. To relate our monadic function back to small-step abstract state machines we introduce another interface which all monads in the framework must obey:

```
MonadStateSpace m :=
  ss : Type -> Type
  transition : ∀ A B, (A -> m B) -> ss A -> ss B
MonadStateSpaceFunctor t :=
  mapMonadStateSpace : ∀ m, MonadStateSpace m -> MonadStateSpace (t m)
```

and we prove the following lemmas:

```
∀ S, MonadStateSpaceFunctor (StateT S) where
  ss (StateT S m) A := ss_m (A × S)

MonadStateSpaceFunctor SetT where
  ss (SetT m) A := ss_m (Set A)
```

[transition] establishes a relationship between the space of actions in the monad and the (pure) space of transitions on an abstract machine state space. If **m** is shown to implement MonadStateSpace, then the semantics of our language can be described as the least fixed point of iterating [transition step] from e, i.e.: $[\![e]\!] = \mu\, x \to e \sqcup \text{transition}\; x$

To recover the various semantics we are interested in, the monad **m** can then be instantiated with following monad transformer stacks:

```
  m1      := EnvStateT (Env Symbol (Set AInt)) Set
  ss m1 A ≈ Set (A × Env Symbol (Set AInt))
  m2      := SetT (EnvState (Env Symbol (Set AInt)))
  ss m2 A ≈ Set A × Env Symbol (Set AInt)
```

and inc functions:

```
inc-concrete : AInt -> Set AInt
inc-concrete (Exact n) = return (Exact (n + 1))
inc-concrete (Symbolic IsZero) = return (Symbolic IsPos)
inc-concrete (Symbolic IsNeg) = return (Symbolic IsNeg)
                                <+> return (Symbolic IsZero)
inc-concrete (Symbolic IsPos) = return (Symbolic IsPos)

inc-abstract : AInt -> AInt -> Set AInt
inc-abstract (Exact n) | n < -1  = return IsNeg
                       | n == -1 = return IsZero
                       | n > -1  = return IsPos
inc-abstract (Symbolic IsZero) = return (Symbolic IsPos)
inc-abstract (Symbolic IsNeg) = return (Symbolic IsNeg)
                                <+> return (Symbolic IsZero)
inc-abstract (Symbolic IsPos) = return (Symbolic IsPos)
```

When instantiated with ⟨**m1, inc-concrete**⟩ a *concrete* interpreter is recovered. When instantiated with ⟨**m1, inc-abstract**⟩ a computable *abstract* interpreter is recovered. When instantiated with ⟨**m2, inc-abstract**⟩ a computable *abstract* interpreter with *heap widening* is recovered.

5 Correctness of the approach

Novel in this work is the justification that a monadic approach to abstract interpretation can produce sound and complete abstract interpreters by construction. As is traditional in abstract interpretation, we capture the soundness and correctness of derived abstract interpreters by establishing a Galois connections between concrete and abstract state spaces, and by proving an order relationship between concrete and abstract transition functions.

We remind the reader of the definition of Galois connection:

```
Galois (A B : Type) :=
  α : A ↗ B
  γ : B ↗ B
  inverses : α∘γ ⊑ id ⊑ γ∘α
```

where $A \nearrow B$ is written to mean the *monotonic* function space between A and B, namely [f : A ↗ B] means [∃ f' : A → B ∀ x y, x ⊑ y → f x ⊑ f y].

Next we introducing the concept of a *functorial Galois connection*, which establishes a Galois connection between functors:

```
FunctorialGalois (m₁ m₂: Type -> Type) :=
  mapGalois : Galois A B -> Galois (m₁ A) (m₂ B)
GaloisTransformer (t : (Type -> Type) -> Type -> Type) :=
  liftGalois : FunctorialGalois m₁ m₂ -> FunctorialGalois (t m₁) (t m₂)
```

and we say (m : Type → Type) is a GaloisFunctor if it is proper in the FunctorialGalois relation:

```
GaloisFunctor m := FunctorialGalois m m
```

We prove the following lemmas:

```
GaloisTransformer (StateT S)
GaloisTransformer SetT
GaloisFunctor Set
```

For a language L and two monads m_1 and m_2, the concrete and abstract state spaces in such a setting are $ss_{m_1} L$ and $ss_{m_2} L$, and the step functions are (transition $step_{m_1}$) and (transition $step_{m_2}$). To establish the relation (transition $step_{m_1}$) $\sqsubseteq$ (transition $step_{m_2}$) we first enrich the (MonadStateSpace m) predicate to establish an isomorphism between m and ss_m:

```
MonadStateSpace m :=
  ss : Type -> Type
  transition : ∀ A B, Iso (A ↗ m B) (ss A ↗ ss B)
```

where

```
Iso A B :=
  to : A → B
  from : B → A
  inverses : to ∘ from = id = from ∘ to
```

Given this stronger connection between monadic actions and their induced state space transitions, we can now prove:

```
FunctorialGalois m₁ m₂
MonadStateSpace m₁
MonadStateSpace m₂
step : ∀ m, L ↗ m L
─────────────────────────────────
Galois (ss_m₁ L) (ss_m₂ L)
transition_m₁ step : ss_m₁ L → ss_m₁ L
transition_m₂ step : ss_m₂ L → ss_m₂ L
transition_m₁ step ⊑ transition_m₂ step
```

which establishs the Galois connection between both the concrete and abstract state spaces and interpreters interpreters induced by m_1 and m_2. [... fancy diagram that sketches the proof ...]

Going back to our example, the only thing left to prove is that our step function is monotonic. Because **step** is completely generic to an underlying monad, we must enrich **all** interfaces that we introduced in Section ? to carry monotonic functions. Generalizing these interfaces, proving that the suppliers of the interfaces (the monad transformers) meet the monotonicity requirements, and proving that the client of the interfaces (the step function) are all monotonic is a systematic exercize in applying the functionality of the $\_\sqsubseteq\_$ relation. We automate the proofs of such theorems using the Coq proof assistant (see Appendix). However we note that such a property can be shown to hold by construction for all morphisms in an embedded logic (as can be done for equality) and therefore it should come as no surprise that if all arrows are changed from $\rightarrow$ to $\nearrow$ then everything "just works out".

Once it is established that our step function is monotonic, the proof burden remains to establish [inc-concrete $\sqsubseteq$ inc-abstract]. [...]

We give a proof that heap widening is sound and less precise than heap cloning by proving [FunctorialGalois (StateT S Set) (SetT (State S))]. [...]

Now we have an end-to-end proof that:

transition $\text{step}_{\langle \mathrm{m_1}, \text{inc-concrete} \rangle}$
$\sqsubseteq$
transition $\text{step}_{\langle \mathrm{m_1}, \text{inc-abstract} \rangle}$
$\sqsubseteq$
transition $\text{step}_{\langle \mathrm{m_2}, \text{inc-abstract} \rangle}$

or

```
⟨concrete semantics⟩
⊑
⟨abstract semantics⟩
⊑
⟨abstract semantics w/heap widening⟩
```

Proving this uses the facts [FunctorialGalois m_1 m_2, MonadStateSpace m_1, MonadStateSpace m_2]–facts we get for free from the framework through composition of monad transformers–and [inc-concrete $\sqsubseteq$ inc-abstract]–a proof obligation of the user of the framework.

6 Expressiveness and modularity

The Abstracting Abstract Interpreters (AAI) framework developed by Might and Van Horne provides an axis on which to develop a spectrum of analyses of varying precision and intensional behavior. The key benefit of their approach is that it generalizes to arbitrary langauges and their state spaces.

This framework provides a separate orthogonal axis on which extensions to abstract interpretation can be explained, and is similarly modular and language agnostic. In particular, the techniques of AAI and Monadic Abstract Interpretation can be combined arbitrarily.

7 Related Work

8 Future Work