A Modular Monadic Framework for Abstract Interpretation (Draft)

David Darais

April 8, 2014

Abstract

Calculational approaches to abstract interpretation–pioneered by Cousot and Cousot[?]–enable the working language designer to systematically extract abstract interpreters induced by Galois connections. Despite the key property that calculated interpreters are correct by construction, the approach falls short in practice; the effort required to calculate often outweighs the correctness payoff. We present a reformulation of monadic abstract interpretation–a computational framework for abstract interpretation introduced by Sergey et al.[?]–which captures a large class of both automatically derivable and correct by construction abstract interpreters. We achieve this by constructing abstract interpreters and Galois connections side-by-side and in compositional fragments. Key players in our framework are a restricted class of monad transformers which are also Galois functor transformers, and whose monadic actions are isomorphic to transitions in a pure (non-monadic) state space.

1 Introduction

1.1 Basic Order Theory

[TODO: Partial Orders. Define "discrete ordering".]
[TODO: Products.]
[TODO: Function Spaces.]
[TODO: Sets.]
[TODO: Monotonicity.]
[TODO: Fixpoints.]

1.2 Basic Abstract Interpretation Concepts

[TODO: Small-step semantics.]
[TODO: Galois Connections.]

[TODO: An extremely simple example (borrow this from Cousot).]
[TODO: A calculational derivation of the previous example.]

2 Monadic Abstract Interpretation By Example

To demonstrate our monadic abstract interpretation framework we guide the reader through the design and implementation of 0CFA for a CPS language with small-step semantics driven by three separate methodologies:

- Ad-hoc design and "a posteriori" verification
- Calculation a la Cousot and Cousot
- Using our monadic framework

General computational, correctness and modularity properties of the framework are deferred to sections 3, 4 and 5. 0CFA is chosen because it is well studied, challenging, and useful for analysis of higher order languages. CPS is chosen because it is in some sense the simplest language for which 0CFA is meaningful. We will use Haskell as both an implementation language and semantics meta-language.

2.1 Concrete Semantics

Figure 1 shows the syntax and semantics of CPS in the form of a small-step interpreter written in Haskell. [TODO: what is call, atomic, op, etc–why the language factored like it is.] Notable features include branching, arithmetic, and a notion of types (branching on an integer is not valid). We assume the discrete equality and ordering relations for CPS and conclude that **step** is monotonic by inspection.

2.2 Ad-hoc Design and "A Posteriori" Verification

In the "a posteriori" approach to abstract interpretation the analysis designer invents the computational content of the desired analysis and then attempts to establish soundness and (less essential) tightness properties in an ad-hoc fashion. This typically consists of forming a Galois connection between concrete and abstract state spaces, and showing either ($\alpha(\text{step}) \sqsubseteq \widehat{\text{step}}$) or ($\text{step} \sqsubseteq \gamma(\widehat{\text{step}})$) (they are equivalent). This approach is representative of how (sound) static analyses are generally justified in practice.

The abstract state space and step function for 0CFA are shown in figure 2. Several seemingly arbitrary choices were made in the design of the abstract state space shown. Once this choice is made and the Galois connection relating it to the concrete state space is established, the abstract step function, and therefore the entire static analysis, is uniquely uniquely determined. Exploiting this uniqueness to *derive* rather than *verify* the abstract step relation underlies the calculational approach, which is demonstrated in the next section. For now,

```
data Lit = I Integer | B Bool
data Op = Add1 | Sub1 | IsNonNeg
data Atom =                          data Call =
    LitA Lit                             IfC Atom Call Call
  | VarA String                        | AppC Atom Atom Atom
  | PrimA Op Atom                      | RetC Atom Atom
  | LamA String String Call            | HaltC Atom
  | KonA String Call
type Env = Map String Val
data Val = LitV Lit | CloV [String] Call Env

op :: Op -> Val -> Maybe Val
op Add1 (IntV n) = Just (IntV n)
op Sub1 (IntV n) = Just (IntV n)
op IsNonNeg (IntV n) | n >= 0 = Just (BoolV True)
                     | otherwise = Just (BoolV False)
op _ _ = Nothing

atomic :: Atom -> Env -> Maybe Val
atomic (LitA l) _ = Just (LitV l)
atomic (VarA x) e = Map.lookup x e
atomic (PrimA o a) _ = case atomic a of
  Just v -> op o v
  Nothing -> Nothing
atomic (LamA x kx c) e = Just (CloV [x, kx] c e)
atomic (KonA x c) e = Just (CloV [x] c e)

call :: Call -> Env -> Maybe (Call, Env)
call (IfC a tc fc) e = case atomic a of
  Just (BoolV True) -> Just (tc, e)
  Just (BoolV False) -> Just (fb, e)
  _ -> Nothing
call (AppC fa xa ka) e = case (atomic fa e, atomic xa e, atomic ka e) of
  (Just (CloV [x, kx] c e'), Just xv, Just kv) ->
    Just (c, Map.insert x xv (Map.insert kx kv e'))
  _ -> Nothing
call (RetC ka xa) e = case (atomic ka, atomic xa) of
  (Just (CloV [x] c e', Just xv) -> Just (c, Map.insert x xv e')
  _ -> Nothing
```

Figure 1: Concrete Semantics for CPS

demonstrating a Galois connection (Maybe (Call, Env)) $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ (Set Call, AEnv), showing that **astep** is monotonic, and showing that (α(step) $\sqsubseteq$ $\widehat{\text{step}}$) are all proof burdens for the analysis designer. We do not detail such proofs here, rather we refer the reader to [Shivers] and [Might].

We make three observations in the relationship between the concrete and abstract semantics thus far.

- The choice of abstract state space was in some sense arbitrary. For example, we could have chosen (Set (Call, Env)), yielding a flow-sensitive analysis.
- **acall** would look much prettier if written in monadic style. This observation was the initial inspiration for monadic abstract interpretation; we'll see how that story plays out in a later section.
- Modifications to either the concrete or abstract semantics will likely render the other incompatible.

We argue this last point is the primary point of pain in designing abstract interpreters. For a large real-world language, the semantics may subtly change during development. Even more likely, the analysis will be augmented and improved over time, as new questions wish to be asked of the analysis. Maintaining each

```
type AEnv = Map String (Set AVal
data AVal = IntAV | BoolAV | CloAV [String] Call

aop :: Op -> AVal -> Set AVal
aop (Add1 IntAV) = Set.singleton IntAV
aop (Sub1 IntAV) = Set.singleton IntAV
aop (IsNonNeg IntAV) = Set.singleton BoolAV
aop _ = Set.empty

aatomic :: Atom -> AEnv -> Set AVal
aatomic (LitA (IntL _)) _ = Set.singleton IntAV
aatomic (LitA (BoolL )) _ = Set.singleton BoolAV
aatomic (VarA x) e = case Map.lookup x e of
  Just avs -> avs
  Nothing -> Set.empty
aatomic (PrimA o a) _ = case aatomic a of
  Just v -> aop o v
  Nothing -> Set.empty
aatomic (LamA x kx c) e = Set.singleton (CloAV [x, kx] c)
aatomic (KonA x c) e = Set.singleton (CloAV [x] c e)

acall :: Call -> Env -> (Set Call, AEnv)
acall (IfC a tc fc) e =
  if BoolAV `elem` aatomic a
    then (Set.fromList [ tc, fb ], e)
    else (Set.empty, e)
acall (AppC fa xa ka) e =
  merge (Set.empty, e) (concatMap (\ fv -> case fv of
    CloAV [x, kx] c ->
      let e' = Map.unionWith Set.union (Map.singleton x (aatomic xv e))
          e'' = Map.unionWith Set.union (Map.singleton kx (aatomic kx e'))
      in [(c, e'')]
    _ -> []) (Set.toList (aatomic fa)))
  where
    merge (cs, e) [] = (cs, e)
    merge (cs, e) (c, e') = (Set.insert c cs, join e e')

acall (RetC ka xa) e =
  Set.unions (Set.map (\ fv -> case fv of
    CloAV [x] c ->
      let e' = Map.unionWith Set.union (Map.singleton x (aatomic xv e))
      in Set.singleton (c, e')
    _ -> Set.empty)) (aatomic fa)
```

Figure 2: "A Posteriori" Abstract Semantics for CPS

of the semantics (and their proofs) to agree on modifications is time consuming and error prone. It is precisely this use case–the continuing evolution of a concrete and/or abstract semantics–where the analysis designer greatly benefit from using our framework.

2.3 Calculational Derivation

In the calculational approach to abstract interpretation, the analysis designer begins by designing a Galois connection $SS \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{SS}$, where SS and $\widehat{SS}$ are the concrete and abstract state spaces. The maps α and γ *induce* an abstract step transition through pre and post-composition:

$$\widehat{\text{step}} = \alpha \circ \text{step} \circ \gamma$$

However, this naive definition of $\widehat{\text{step}}$ isn't constructive; γ will not be computable in general. $\widehat{\text{step}}$ must be refined in a calculational style, which takes the form of a chain of rewrites, after which a computable function is revealed.

2.4 Monadic Derivation

```
type MEnv addr = Map String addr
type MStore addr = Map addr (Set MVal)
data MVal addr = LitMV Lit | IntMV | BoolMV | CloMV [String] Call (MEnv addr)

type Analysis m =
  ( Monad m
  , MonadPlus m
  , MonadState (MEnv addr)
  , MonadState (MStore addr)
  )

matomic :: (Lit -> Set MVal) -> (Op -> MVal -> Set MVal) -> Atom -> m (Set MVal)
matomic lit _ (LitA n) = return (lit n)
matomic _ _ (VarA x) = do
  e <- getEnv
  case Map.lookup x e of
    Nothing -> mzero
    Just vs -> return vs
matomic lit op (PrimA o a) _ =
  vs <- matomic lit op a
  v <- join vs
  return (op o v)
matomic _ _ (LamA x kx c) = do
  e <- getEnv
  return (Set.singleton (CloMV [x, kx] c e))
matomic _ _ (KonA x c) = do
  e <- getEnv
  return (Set.singleton (CloMV [x] c e))

mcall :: (Lit -> Set MVal) -> (Op -> MVal -> Set MVal) -> Call -> m Call
mcall lit op (IfC a tc fc) = do
  v <- matomic lit op a
  case v of
    LitMV (BoolL True) -> return tc
    LitMV (BoolL False) -> return tc
    BoolMV -> return tc <+> return fc
    _ -> mzero
mcall lit op (AppC fa xa ka) =
  fvs <- matomic lit op f
  fv <- join fv
  case fv of
    CloMV [x, k] c e -> do
      xvs <- matomic lit op xa
      kvs <- matomic lit op ka
      putEnv (Map.insert x xvs (Map.insert k kvs e))
      return c
    _ -> mzero
mcall lit op (RetC ka xa) =
  kvs <- matomic lit op ka
  kv <- join kvs
  case kv of
    CloMV [x] c e -> do
      xvs <- matomic lit op xa
      putEnv (Map.insert x xvs e)
      return c
    _ -> mzero
\caption{Abstract CPS Language and Semantics}
\label{cps_monadic}
```

Monadic abstract interpretation was introduced by Sergey et al. as a method for simultaneously defining multiple versions of a semantics in one go. The primary contribution of this work is a proof framework to augment this approach, showing that the derived abstract interpreters are correct by construction. For now we contrast the burdon of proof for the analysis designer with the a posteriori verification described in the previous section.

First we translate the semantics of CPS into monadic form as shown in figure ?, being careful to leave the semantics general enough to capture both concrete and abstract semantics. We note that certain design decisions must be made up front, like what the abstract value space looks like, but defer the full choice of state space to a particular instantiation within the monadic framework Independent of other languages, we need only justify that mstep is monotonic, which is true by simple inspection.

We now recover both the concrete semantics and multiple abstract semantics from the monadic semantics by:

- Designing delta and literal semantic functions

- Building an instance of the abstract monadic effect interface

3 Computational Framework for Monadic AI

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 <+> ...
```

3.1 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.

4 Correctness Framework for Monadic AI

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

5 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 languages 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.

6 Related Work

7 Future Work