A Modular Monadic Framework for Abstract Interpretation (Draft)

David Darais

August 4, 2014

Abstract

The design and implementation of static analyses is a difficult process. Verifying the correctness of an implementation is often the most painful step. We present Monadic AAM–an extension of the Abstracting Abstract Machines methodology introduced by Van Horn and Might [1]–which captures a large class of both automatically derivable and correct by construction abstract interpreters. Monadic AAM is part methodology and part toolkit. In the methodology, semantics are designed in a monadic extension of traditional AAM. Once designed, a large class of known analyses can be automatically recovered using our language agnostic toolkit. Both the computational and correctness properties of our framework are realized through a restricted class of monad transformers. Our framework enjoys the benefits of being highly compositional and placing a minimal burden of proof on the analysis designer.

Contents

1 Introduction

2 Background

3 AAM By Example

3.1 CPS

The state space for CPS syntactically separates `Call` expressions from `Atom` expressions.

```
type Name = String
data Lit = I Integer | B Bool
data Op = Add1 | Sub1 | IsNonNeg
data Atom = LitA Lit | Var Name | Prim Op Atom | Lam [Name] Call
data Call = If Atom Call Call | App Atom [Atom] | Halt Atom
```

`Call` expressions are separated from `Atom` expressions because they may not terminate, and therefore cannot be given a striaghtforward denotation. Each syntactic form in `Call` is designed to be recursive in only one position, eliminating the need for a call stack when we design the operational semantics.

`Atom` expressions are exactly those for which we can compute a denotation. The `Lam` syntactic form may look troubling because it is mutually recursive with `Call`, but the denotation of `Lam xs c` will be a closure which delays the evaluation of `c` until it is applied in a `Call` rule.

Possibly non-standard additions to our CPS language are conditional branching (`IfC`); two literal types, integer and boolean; and primitive operations `Add1`, `Sub1` and `IsNonNeg`. We include these to bring CPS closer to a language one might use in practice, and to examine the design space of their abstract semantics.

3.2 Concrete Interpreter

To give `Call` concrete semantics we first design a state space. The state space introduces an environment to track variable bindings, and a value type containing closures (which mutually close over environments).

```
type Env = Map Name Val
data Val = LitV Lit | Clo [Name] Call Env
type StateSpace = Maybe (Call, Env)
```

The state space is partial (uses the `Maybe` type) and `Nothing` is the meaning of failed computations. Computations fail when an expression is ill-typed–for example if a literal flows to function application position–or if a function application is of the wrong arity.

The semantics for `Op` are given denotationally, and are straightforward.

```
op :: Op -> Val -> Maybe Val
op Add1 (LitV (I n)) = Just $ LitV $ I $ n+1
op Sub1 (LitV (I n)) = Just $ LitV $ I $ n-1
op IsNonNeg (LitV (I n)) | n >= 0 = Just $ LitV $ B True
                         | otherwise = Just $ LitV $ B False
op _ _ = Nothing
```

The semantics for `Atom` are given denotationaly.

```
atom :: Env -> Atom -> Maybe Val
atom _ (LitA l) = Just $ LitV l
atom e (Var x) = mapLookup x e
atom e (Prim o a) = case atom e a of
  Nothing -> Nothing
  Just v -> op o v
atom e (Lam xs c) = Just $ Clo xs c e
```

Literals evaluate to themselves. Variables evaluate to a value retrieved from the environment. Lambdas evaluate immediately to closures which capture their environment.

The semantics for `Call` are given *operationally* as a small step function.

```
call :: Env -> Call -> Maybe (Call, Env)
call e (If a tc fc) = case atom e a of
  Just (LitV (B True)) -> Just (tc, e)
  Just (LitV (B False)) -> Just (fc, e)
  _ -> Nothing
call e (App fa xas) =
  case atom e fa of
    Just (Clo xs c e') ->
      case bindMany xs xas e' of
        Nothing -> Nothing
        Just e'' -> Just (c, e'')
    _ -> Nothing
call e (Halt a) = Just (Halt a, e)
```

Conditional statements branch on boolean values. Applications step to a function's body and closure environment with argument values bound to formal parameters. Termination is signaled with the `Halt` command. Helper function `bindMany xs xas e` evaluates the function arguments `xas` and binds them to the formal parameters `xs` in the environment `e`.

```
bindMany :: [Name] -> [Atom] -> Env -> Maybe Env
bindMany [] [] e = Just e
bindMany (x:xs) (xa:xas) e =
  case (atom xa e, bindMany xs xas e) of
    (Just xv, Just e') -> Just (mapInsert x xv e')
    _ -> Nothing
bindMany _ _ _ = Nothing
```

`bindMany xs xas e` fails if evaluating any argument in `xas` fails or if there is an arity mismatch.

The full semantics of `Call` are given by the transitive closure of `step`.

```
step :: StateSpace -> StateSpace
step Nothing = Nothing
step (Just (c, e)) = call e c

exec :: Call -> StateSpace
exec c0 = iter step $ Just (c0, mapEmpty)
```

3.3 State Space Abstraction

To arrive at a computable analysis we must abstract the infinite concrete state space with a finite abstract state space. As we sketch the abstract state space

in the AAM methodology we will hint at corrosponding changes that need to take place in the semantics. The full abstract semantics for the final abstract state space will be given in section 3.4.

3.3.1 Cutting Recursion

In the AAM methodology, we first identify recursion in the state space and replace it with indirection through addresses and a store.

```
type Addr = Integer
type Time = Integer
type Env = Map Name Addr
type Store = Map Addr Val
data Val = LitV Lit | Clo [Name] Call Env
type StateSpace = Maybe (Call, Env, Store, Time)
```

Cutting Recursion

Abstract time is introduced into the state space as a constantly increasing value from which to allocate fresh addresses.

As a result of this change in the state space, the semantics will need to change accordingly. Variable lookup must fetch an address from the environment, and then fetch the corresponding value bound to that address from the store. Variable binding must allocate globally fresh addresses from the current time, binding the formal parameter to the address in the environment and the address to the value in the store.

3.3.2 Address and Time Abstraction

Now that recursion is eliminated from the state space we take aim at unbounded parts of the state space, with the addresses and time that we just introduced being the first to go. Rather than make a single choice of abstract address and time, we leave this choice as a parameter. All that is required is:

- a time `tzero` to begin execution
- a function `tick` which moves time forward
- a function `alloc` which allocates an address for binding a formal parameter at the current time

We encode this interface using a Haskell type class `AAM` with associated types `Addr` and `Time`.

```haskell
class AAM aam where
  type Time (aam :: *) :: *
  type Addr (aam :: *) :: *
  tzero :: aam -> Time aam
  tick :: aam -> Call -> Time aam -> Time aam
  alloc :: aam -> Name -> Time aam -> Addr aam

type Env aam = Map Name (Addr aam)
type Store aam = Map (Addr aam) (Val aam)
data Val aam = LitV Lit | Clo [Name] Call (Env aam)
type StateSpace aam = Maybe (Call, Env aam, Store aam, Time aam)
```

Address and Time Abstraction

An implementation of `AAM aam0` for a particular type `aam0` must pick two types, `Addr aam0` and `Time aam0`, and then implement `tzero`, `tick`, `alloc`, for those types. The type `aam0` its self will only used as a type level token for selecting associated types at the type level, and will have a single inhabitant for selecting type class functions at the value level. (If Haskell had a proper module system this would all be better expressed as a module interface–containing a type and functions over that type–rather than an associated type, type class and proxy singleton type.)

3.3.3 Introducing Nondeterminism

Following the AAM methodology, we realize that in a world with finite adresses but potentially infinite executions, there is a chance we will need to reuse an existing address. This causes us to introduce a powerset both around the state space and into the domain of the heap, as a single address may now point to multiple values, and variable reference may return multiple values.

```haskell
type Store aam = Map (Addr aam) (Set (Val aam))
type StateSpace aam = Set (Call, Env aam, Store aam, Time aam)
```

Introducing Nondeterminism

As a result of this abstraction, the semantics must consider multiple control paths where before there was only one.

3.3.4 Delta Abstraction

Now we have a *nondterministic* semantics paramterized by abstract addresses–but we are not yet finished. The state space is still infinite due to the unboundedness of integer literal values. (Even if we took the semantics of finite machine integers, 2^{32} is still much too large a state space to be exploring in a practical

analysis.) To curb this we introduce yet another axis of parameterization with the intent of recovering multiple analyses from the choice later. In the Haskell implementation we represent this axis with the type class `Delta` and associated type `Val`.

```
class AAM aam where
  type Addr (aam :: *) :: *
  type Time (aam :: *) :: *
  tzero :: aam -> Time aam
  tick :: aam -> Call -> Time aam -> Time aam
  alloc :: aam -> Name -> Time aam -> Addr aam
class Delta d where
  type Val (d :: *) :: * -> *
  lit :: d -> Lit -> Val d aam
  clo :: d -> [Name] -> Call -> Env aam -> Val d aam
  op :: d -> Op -> Val d aam -> Maybe (Val d aam)
  elimBool :: d -> Val d aam -> Set Bool
  elimClo :: d -> Val d aam -> Set ([Name], Call, Env aam)
type Env aam = Map Name (Addr aam)
type Store d aam = Map (Addr aam) (Set (Val d aam))
type StateSpace d aam = Set (Call, Env aam, Store d aam, Time aam)
```

(Final) Delta Abstraction

Different analyses may want to make different choices in their value representations, so we don't commit to a particular `Val` type, rather we require that *some* `Val` type exist with a particular interface. All that matters is that we have *introduction* forms for literals and closures, a *denotation function* for primitive operations, and *elimination* forms for booleans and closures. Eliminators are only required for booleans and closures because those are the only values scrutenized in the control flow of the semantics.

3.4 Abstract Semantics

Now that we have designed an abstract state space–albeit highly parameterized–we must turn to implementing its semantics. The goal in this entire exercise is to develop the abstract state space and semantics in such a way that concrete and abstract interpreters can be derived from a single implementation.

Following the structure of the concrete semantics, we adapt each function to the new abstract state space. `op` is now entirely implemented by the designer of the analysis as part of the `Delta` interface. `atom` must follow another level of indirection in variable lookups, and must account for nondeterminism.

```
atom :: (Delta d) => d
     -> Env aam -> Store d aam -> Atom -> Set (Val d aam)
atom d _ _ (LitA l) = setSingleton $ lit d l
```

```
atom _ e s (Var x) = case mapLookup x e of
  Nothing -> setEmpty
  Just l -> mapLookup l s
atom d e s (Prim o a) = eachInSet (atom d e s a) $ \ v ->
  case op d o v of
    Nothing -> setEmpty
    Just v' -> setSingleton v'
atom d e _ (Lam xs c) = setSingleton $ clo d xs c e
```

`call` must use the eliminators provided by `Delta d` to guide control flow.

```
call :: (Delta d, AAM aam) => d -> aam
     -> Time aam -> Env aam -> Store d aam -> Call
     -> Set (Call, Env aam, Store d aam)
call d aam t e s (If a tc fc) =
  eachInSet (atom d e s a) $ \ v ->
    eachInSet (elimBool d v) $ \ b ->
      setSingleton (if b then tc else fc, e, s)
call d aam t e s (App fa xas) =
  eachInSet (atom d e s fa) $ \ v ->
    eachInSet (elimClo d v) $ \ (xs,c,e') ->
      setFromMaybe $ bindMany aam t xs xas e' s
call _ _ _ e s (Halt a) = setSingleton (Halt a, e, s)
```

`bindMany` must join stores when binding to addresses, in case the address is already in use.

```
bindMany :: (AAM aam) => aam
         -> Time aam -> [Name] -> [Atom]
         -> Env aam -> Store d aam
         -> Maybe (Env aam, Store d aam)
bindMany _   _ [] [] e s = Just (e, s)
bindMany aam t (x:xs) (xa:xas) e s = case bindMany aam t xs xas e s of
  Nothing -> Nothing
  Just (e', s') ->
    let l = alloc aam x t
    in Just (mapInsert x l e', mapInsertWith (\/) l xv s')
bindMany _ _ _ _ _ _ = Nothing
```

Finally, `step` must advance time as it advances each `Call` state, and `exec` is modified to a collecting semantics.

```
step :: (Delta d, AAM aam) => d -> aam
     -> StateSpace d aam -> StatSpace d aam
step d aam ss = eachInSet ss $ \ (c,e,s,t) ->
  eachInSet (call d aam t c e s) $ \ (c',e',s') ->
    setSingleton (c',e',s',tick aam c' t)
```

```
exec :: (Delta d, AAM aam) => d -> aam
     -> Call -> StateSpace d aam
exec d aam c0 = iter (collect (step d aam)) $
  setSingleton (c0, mapEmpty, mapEmpty, tzero aam)
```

The collecting semantics keeps track of all previously seen states. Keeping track of previous states turns exec into a monotonic function, which means its iteration will terminate if the instantiated state space is finite.

3.5 Recovering the Concrete Interpreter

The final delta abstraction resulted in a highly factorized state space, amenable to a wide range of analyses. Before manipulating these axes of parameterization, we first show how to recover a concrete interpreter.

Our semantics has two axes of parameterization: `AAM` and `Delta`. For `AAM` we choose integers for abstract time and integers paired with variable names for abstract addresses.

```
data C_AAM = C_AAM
instance AAM C_AAM where
  type Time C_AAM = Integer
  type Addr C_AAM = (Integer, Name)
  tzero C_AAM = 0
  tick C_AAM _ t = t+1
  alloc C_AAM x t = (t, x)
```

For `Delta` we choose the value type and primitive operations from section 3.2.

```
data C_Delta = C_Delta
data ConcreteVal aam = LitV Lit | Clo [Name] Call (Env aam)
instance Delta C_Delta where
  type Val C_Delta = ConcreteVal
  lit C_Delta l = Lit l
  clo C_Delta xs c e = Clo xs c e
  op C_Delta Add1 (LitV (I i)) = Just $ LitV $ I $ i+1
  op C_Delta Sub1 (LitV (I i)) = Just $ LitV $ I $ i-1
  op C_Delta IsNonNeg (LitV (I i))
    | i >= 0 = Just $ LitV $ B $ True
    | otherwise = Just $ LitV $ B $ False
  op C_Delta _ _ = Nothing
  elimBool C_Delta (LitV (B b)) = setSingleton b
  elimBool C_Delta _ = setEmpty
  elimClo C_Delta (Clo xs c e) = setSingleton (xs, c, e)
  elimClo C_Delta _ = setEmpty
```

Running the abstract semantics from section 3.4 with these two parameterizations–`C_AAM` and `C_Delta`–recovers the concrete (collecting) semantics.

```
execConcrete :: Call -> StateSpace C_Delta C_AAM
execConcrete = exec C_Delta C_AAM
```

3.6 Recovering 0CFA

To recover 0CFA we use variables themselves as abstract addresses and ignore the time component of the state space.

```
data ZCFA_AAM = ZCFA_AAM
instance AAM ZCFA_AAM where
  type Time ZCFA_AAM = ()
  type Addr ZCFA_AAM = Name
  tzero ZCFA_AAM = ()
  tick ZCFA_AAM _ () = ()
  alloc ZCFA_AAM x () = x
```

Although the `ZCFA_AAM` parameter captures the behavior of 0CFA, we do not yet have a computable analysis–we need a finite instantiation for the `Delta` parameter. We pick a simple `Delta` instance for now, but note that it is not inherent to 0CFA, it merely serves as *some* abstraction for literals and primitive operations.

```
data Sym_Delta = Sym_Delta
data SymVal aam = IntV | BoolV | Clo [Name] Call (Env aam)
instance Delta Sym_Delta where
  type Val Sym_Delta = SymVal
  lit Sym_Delta (I _) = IntV
  lit Sym_Delta (B _) = BoolV
  clo Sym_Delta xs c e = Clo xs c e
  op Sym_Delta Add1 IntV = Just IntV
  op Sym_Delta Sub1 IntV = Just IntV
  op Sym_Delta IsNonNeg IntV = Just BoolV
  op Sym_Delta _ _ = Nothing
  elimBool Sym_Delta BoolV = setSingleton True \/ setSingleton False
  elimBool Sym_Delta _ = setEmpty
  elimClo Sym_Delta (Clo xs c e) = setSingleton (xs, c, e)
  elimClo Sym_Delta _ = setEmpty
```

We call this `Sym_Delta` because it treats literals purely symbolically, essentially carrying around only their type information.

Running the abstract semantics from section 3.4 with these two parameterizations–`ZCFA_AAM` and `Sym_Delta`–recovers a 0CFA analysis.

```
execSymZCFA :: Call -> StateSpace Sym_Delta ZCFA_AAM
execSymZCFA = exec Sym_Delta ZCFA_AAM
```

3.7 Recovering kCFA

To recover kCFA we record a finite number of calling contexts as abstract time, and pair these with names for addresses.

```
data kCFA_AAM = kCFA_AAM Int
instance AAM kCFA_AAM where
  type Time kCFA_AAM = [Call]
  type Addr kCFA_AAM = (Name, [Call])
  tzero (kCFA_AAM k) = []
  tick (kCFA_AAM k) c t = take k (c:t)
  alloc (kCFA_AAM k) x t = (x, t)
```

Running the abstract semantics from section 3.4 with this parameterization and the symbolic delta parameter–ZCFA_AAM and Sym_Delta–recovers a kCFA analysis.

```
execSymZCFA :: Call -> StateSpace Sym_Delta ZCFA_AAM
execSymZCFA = exec Sym_Delta ZCFA_AAM
```

3.8 Optimizations

We now introduce two optimizations to AAM–heap widening and abstract garbage collection–and show how they integrate into the computational framework described thus far.

3.8.1 Heap Widening

The state space we arrived at from our methodological abstraction of the concrete state space in section 3.3 was

```
StateSpace d aam = Set (Call, Env aam, Store d aam, Time aam)
```

This state space will lead to a fully flow sensitive analysis, and consequently an exponential runtime. The fix is well known in the literature as *heap widening*, where the desired state space is

```
StateSpace d aam = (Set (Call, Env aam, Time aam), Store d aam).
```

When widening the heap, each store in the set of states is joined together to form a single universally approximating store. This results in a less precise but vastly more efficient abstraction.

Introducing heap widening into the analysis can be accomplished in two ways: rewriting all the semantics for the new state space, or by introducing a post-facto widening operation that operates on the original state space. We demonstrate the latter approach, which can be understood as a Galois connection between the inefficient and efficient state spaces.

$$\texttt{Set (Call, Env, Store, Time)} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \texttt{(Set (Call, Env, Time), Store)}$$

We show only the α portion of the Galois connection and call it `widen`.

```
widen :: StateSpace d aam -> StateSpace d aam
widen ss =
```

```
  let wideStore = setJoins $ setMap (\ (_,_,s,_) -> s) ss
  in setMap (\ (c,e,_,t) -> (c,e,wideStore,t))

wideStep :: (Delta d, AAM aam) => d -> aam
         -> StateSpace d aam -> StateSpace d aam
wideStep d aam = widen . step d aam

widenExec :: (Delta d, AAM aam) => d -> aam
          -> Call -> StateSpace d aam
widenExec d aam c0 = iter (collect (wideStep d aam)) $
  setSingleton (c0, mapEmpty, mapEmpty, tzero aam)
```

3.8.2 Abstract Garbage Collection

Abstract garbage collection has become standard in structural abstract interpretation and a staple in the AAM methodology. The idea is analogous to garbage collection for memory safe languages. Even though we must accept that abstract addresses can be reused during execution, it would be nice if we could completely replace the contents of an abstract location if we knew no future state would need the old value. This concept–of future states depending on values in the store–can be approximated by the *reachability* of those values in the current state. Abstract garbage collection is the process where unreachable allocations in the store are removed so that future allocations don't require joins in the store.

To perform abstract garbage collection we must examine all addresses in the store and eliminate those which cannot be reached from a *root* in the state space. The roots $\mathcal{T}(c, e)$ in the state space are the locations bound to free variables in c:

$$\mathcal{T}(c, e) = \{e(v) : v \in free(c)\}$$

The reachable addresses $\mathcal{R}(c, e, s, t)$ are those which are reachable from the roots, which is the least solution to the equation:

$$\mathcal{R}(c, e, s, t) = \mathcal{T}(c, e) \cup \{\ell' : \ell \in \mathcal{R}(c, e, s, t), \ell \rightsquigarrow_{\langle e,s \rangle} \ell'\}$$

where $\rightsquigarrow_{\langle e,s \rangle}$ is the single-step points-to relation in environment e and store s. The set of addresses to eliminate is the complement of $\mathcal{R}(c, e, s, t)$

3.8.3 Composing Heap Widening and Abstract GC

Heap widening and abstract garbage collection were introduced independently; what happens when they compose? The result is semantically what you want but exhibits bad behavior operationally. The naive composition of the optimizations results in each store being garbage collected independently of each other, which is expensive. What is worse, *they are all the exact same store*, making the extra effort entirely redundant.

The solution to this is to either rewrite the semantics to use a widened store throughout, or to rewrite abstract garbage collection specifically for the widened

state space. Granted, none of these options are terribly difficult in this setting, but rewriting semantics is a time-consuming and bug-ridden process in general. A more systematic abstraction is clearly needed for these optimizations: one where optimizations compose seamlessly with both one another and the abstract semantics.

4 Monadic AAM

We now present monadic AAM. Monadic AAM is part an extension of the AAM methodology, and part extension of the AAM toolkit. In methodology, interpreters are first abstracted in the AAM tradition, and then further factored to be monadic. In toolkit, several monads are provided by our library to recover various analysis properties in a language agnostic fashion.

4.1 AAM in Monadic Style

Monadic AAM begins with recognizing that the abstract semantics from section 3.4 look suspiciously monadic. In this section we will rewrite the exact same function but in monadic style. After the exercise, the monad will be generalized in section **??** and this generalization will be exploited in section **??**.

4.1.1 Nondeterminism

Before introducing the nondeterminism monad, we first describe its interface, `MonadPlus`. `MonadPlus` abstractly encodes the notion of nondeterminism.

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

`mzero` encodes failure and `mplus` encodes nondeterministic choice.

The nondeterminism monad is built with the list datatype

```
type Nondet = []
instance Monad Nondet where
  return x = [x]
  [] >>= k = []
  (x:xs) >>= k = k x ++ (xs >>= k)
```

and implements the `MonadPlus` interface:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

For example, `[1,3] >>= \ x -> [x, x+1]` and `[1,3] `mplus` [2,4]` both evaluate to `[1,2,3,4]`.

The `atom` denotation function from section **??** can be refactored into the nondeterminism monad by replacing `setSingleton` with `return`, `eachInSet` with `(>>=)`, and `setEmpty` with `mzero`.

```
atom :: (Delta d) => d
     -> Env aam -> Store d aam -> Atom -> Nondet (Val d aam)
atom d _ _ (LitA l) = return $ lit d l
atom _ e s (Var x) = case mapLookup x e of
  Nothing -> mzero
  Just l -> setMSum $ mapLookup l s
atom d e s (Prim o a) = do
  v <- atom d e s a
  mFromMaybe $ op d o v
atom d e _ (Lam xs c) = return $ clo d xs c e
```

`call` can be factored similarly.

```
call :: (Delta d, AAM aam) => d -> aam
     -> Time aam -> Env aam -> Store d aam -> Call
     -> Nondet (Call, Env aam, Store d aam)
call d aam t e s (If a tc fc) = do
  v <- atom d e s a
  b <- setMSum (elimBool d v)
  return (if b then tc else fc, e, s)
call d aam t e s (App fa xas) =
  v <- atom d e s fa
  (xs,c,e') <- setMSum $ elimClo d v
  mFromMaybe $ bindMany aam t xs xas e' s
call _ _ _ e s (Halt a) = return (Halt a, e, s)
```

4.1.2 State

To add state to our monad we apply a state monad transformer on top of the current nondeterminism monad. Monad transformers are compositional building blocks for building larger monads. At the interface level, monad transformers consume monads and produce monads; they are mappings from monads to monads–monad morphisms. Certain monad transformers are known to compose well with others. In our case, nondeterminism and state are known to compose well (in this order only), meaning the resulting monad will implement both state and nondeterminism actions.

As we did with the nondeterminism monad and its interface `MonadPlus`, we introduce an interface for state called `MonadState`. `MonadState s m` means the monad `m` has access to a single cell of state `s`.

The state monad transformer contains a function from state values to a monadic actions of state-result pairs.

```
newtype StateT s m a = StateT { runStateT ::  s -> m (a,s) }
instance (Monad m) => Monad (StateT m) where
```

```
  return x = StateT $ \ s -> return (x, s)
  xM >>= k = StateT $ \ s -> do
    (x,s') <- runStateT xM s
    runStateT (k x) s'
```

and implements the `MonadState` interface

```
instance (Monad m) => MonadState s (StateT s m) where
  get = StateT $ \ s -> return (s,s)
  put s = StateT $ \ _ -> return ((),s)
```

The benefit of monad transformers, and the reason why we bother to declare interfaces for individual monads, is that monad transformer stacks have the ability to expose the interfaces of inner monads to the fully composed outer monad. For example, `Nondet` has been shown to implement `MonadPlus`, but it is also the case that the transformer stack `StateT s NonDet` *also* implements `MonadPlus`. This lifting, or inheritance, of monadic interface can be expressed as a type class instance rule.

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero = StateT $ \ _ -> mzero
  mplus xM yM = StateT $ \ s ->
    runStateT xM s `mplus` runStateT yM s
```

Our state space has *three* state components, but the `MonadState` interface only encodes one cell of state. One solution is to combine each component of the state into a tuple, but this would limit our ability to abstract each component in different ways. A better solution is to allow multiple states in one stack, with a way of selecting which state is desired. The method for selecting the desired state can take many forms: 1) separate type classes could be defined for each state, 2) type singleton tags (like the singletons used to fake modules in section 3) could be defined to select the state, or 3) we could require that each state is of unique type and allow type inference to infer the desired state. We take the approach of 2)–introducing type tags to select the desired state. This approach adds a little more mechanism and type-trickery, but is much less work than the 1) and avoids other newtype "hacks" required to use 3).

We redefine the `MonadState` class to use the `s` class parameter as a tag, rather than the type of the state cell its self. The actual type of the state is an associated type of the tag.

```
class MonadStateTagged s m where
  type Cell s :: *
  get :: s -> m (Cell s)
  put :: s -> Cell s -> m ()
```

The state type changes accordingly to store `Cell s`.

```
newtype StateT s m a = StateT { runStateT ::  Cell s -> m (a,Cell s) }
```

(The Monad, MonadState and MonadPlus instance are analagous.)

Now that we have a tagged state monad, we allow lifting of interior state interfaces to the outside of a transformer stack.

```
instance (MonadState s m) => MonadState s (StateT s' m) where
  get = StateT $ \ s -> do
    s' <- get
    return (s',s)
  put s' = StateT $ \ s -> do
    put s'
    return ((),s)
```

We now rewrite the semantics to use a four-layer monad transformer stack, which we abbreviate to M. The tags for the environment, store and time components of the monad are called E, S and T and defined as type family instances.

```
data E aam = E aam
type instance Cell (E aam) = Env aam
data S d aam = S d aam
type instance Cell (S d aam) = Store d aam
data T aam = T aam
type instance Cell (T aam) = Time aam
type M d aam = StateT (E aam) (StateT (S d aam) (StateT (T aam) []))
```

The atom and call functions are then simplified to monadic actions.

```
atom :: (Delta d) => d -> aam -> Atom -> M d aam (Val d aam)
atom d aam (LitA l) = return $ lit d l
atom d aam (Var x) = do
  e <- get $ E aam
  s <- get $ S d aam
  case mapLookup x e of
    Nothing -> mzero
    Just l -> setMSum $ mapLookup l s
atom d aam (Prim o a) = do
  v <- atom d aam a
  mFromMaybe $ op d o v
atom d aam (Lam xs c) = do
  e <- get $ E aam
  return $ clo d xs c e

call :: (Delta d, AAM aam) => d -> aam -> Call -> M d aam Call
call d aam (If a tc fc) = do
  v <- atom a
  b <- setMSum (elimBool d v)
  return $ if b then tc else fc
call d aam (App fa xas) =
  v <- atom d aam fa
```

```
  (xs,c,e') <- setMSum $ elimClo d v
  mFromMaybe $ bindMany aam xs xas e'
call _ _ (Halt a) = return $ Halt a
```

We are not finished until we also transform `step` to work with our new monadic type. The purpose of `step` is to translate the nondetermistic function `call` into a state space transition. We take this approach to heart and maintain a separation between the monadic type and the type of its state space transitions. Because the state space is separate from the monad, we reuse the same state space from before.

```
type StateSpace d aam = Set (Call, Env aam, Store d aam, Time aam)
```

The `step` function now converts between the monadic action `Call -> M Call` and state space transitions `SateSpace -> StateSpace`.

```
step :: (Delta d, AAM aam) => d -> aam
     -> StateSpace d aam -> StatSpace d aam
step d aam ss = eachInSet ss $ \ (c,e,s,t) -> setFromList $ do
  (((c',e'),s'),t') <-
    flip runStateT e $
    flip runStateT s $
    flip runStateT t $
    call d aam c
  return (c',e',s',tick aam c' t)
```

4.2 Generalizing the Monad

The previous section demonstrated how to write the abstract semnatics from section 3.4 in monadic style. In this section we generalize the monad, allowing us to instantiate to different monad stacks so long as they meet an interface.

The generalized interface will have parameters `d`, `aam` and `m` such that:

- `Delta d` implements a value space primitive operations
- `AAM aam` implements abstract address and time operations
- `m` is a `Monad`, `MonadPlus` and `MonadState` for environment, store and time types.

This interface is encoded in full by the type class constraint `Semantics d aam m`.

```
type Semantics d aam m =
  ( Delta d
  , AAM aam
  , Monad m
  , MonadPlus m
  , MonadState (E aam) m
  , MonadState (S d aam) m
  , MonadState (T aam) m
  )
```

We next manipulate the semantics into a form generic to parameters `d`, `aam` and `m` such that `Semantics d aam m` holds.

```
atom :: (Semantics d aam m) => d -> aam -> Atom -> m (Val d aam)

call :: (Semantics d aam m) => d -> aam -> Call -> m Call
```

The code is omitted because it is identical to the code for the concrete `M` in section **??**

Now we're in some trouble. How do we implement `step` when we don't know what monad we're using? The answer is to require the monad to come with an implementation of `step`. Fortunately `step` can be implemented at the *monad transformer* level, so the implementation can be derived for an arbitrary monad transformer stack.

We capture the ability for a monad to convert monadic actions to state space transitions in another type class: `MonadStep`.

```
class (Pointed (SS m)) => MonadStep m where
  type SS m :: * -> *
  step :: (a -> m b) -> SS m a -> SS m b
```

`MonadStep` has associated type `SS m` for describing the type of step functions `step :: SS m a -> SS m b`. In our framework, we will always be instantiating step to `step :: SS m Call -> SS m Call`. However this polymorphism is crucial to building state space transitions for monad transformers, as they may instantiate the polymorphic type `a` to something else.

We further require that the associated state space `SS m` be `Pointed` so that we can inject pure values into the state space.

```
class Pointed t where
  point :: a -> t a
```

The only monad thus far, nondeterminism, is a `MonadStep` with its state space `SS` being list and `step` being simply monadic bind.

```
instance Pointed [] where
  point = return
instance MonadStep NonDet where
  type SS NonDet = []
  step f xs = xs >>= f
```

The only monad transformer thus far, state, is a `MonadStep` transformer as well, taking `MonadStep` things to `MonadStep` things.

```
newtype StateT_SS s m a = StateT_SS { runStateT_SS :: SS m (a, Cell s) }
instance (Pointed (SS m), JoinLattice (Cell s)) => Pointed (StateT_SS s m) where
  point a = point (a, bot)
instance (MonadStep m) => MonadStep (StateT s m) where
  SS (StateT s m) = StateT_SS s m
  step f xSS = step (\ (a, s) -> runStateT (f a) s) $ runStateT_SS xSS
```

Note that in order for the state space associated with `StateT` to be `Pointed` we require `Cell s` to be a `JoinLattice`, meaning it has a bottom element.

Executing the semantics can be achieved by injecting the initial `Call` value into the state space and iterating the collecting function as before.

```
exec :: (Semantics d aam m) => d -> aam -> Call -> SS m Call
exec d aam c0 = iter (collect (step $ call d aam)) $ point c0
```

4.3 Recovering a Concrete Interpreter

To recover concrete semantics we can reuse the concrete parameters `C_Delta` and `C_AAM`, and need only design a concrete monad to instantiate the semantics.

When implementing concrete semantics in section 3.5 we were forced to reuse the `Set` type even though the semantics were fully deterministic. Because we have abstracted the semantics over a monad, we are now able to choose an appropriately precise domain for execution. A more appropriate domain would be the `Point` type, where the meaning of `Point a` is the extension of `a` with explicit `Top` and `Bot` values.

```
data Point a = Top | Bot | Point a
```

Point can be shown to be a monad:

```
instance Monad Point where
  return = Point
  Top >>= k = Top
  Bot >>= k = Bot
  Point a >>= k = k a
```

and even a monad plus:

```
instance MonadPlus Point where
  mzero = Bot
  mplus Bot yM = yM
  mplus xM Bot = xM
  mplus Top yM = Top
  mplus xM Top = Top
  mplus (Point x) (Point y) = Top
```

and finally a monad step:

```
instance Pointed Point where
  point = Point
instance MonadStep Point where
  type SS Point = Point
  step f xs = xs >>= f
```

Our final monad stack for recovering concrete semantics will be

```
type C_M d aam = StateT (E aam) (StateT (S d aam) (StateT (T aam) Point))
```

and execution of the concrete semantics can be recovered by

```
execConcrete :: Call -> SS (C_M C_Delta C_AAM) Call
execConcrete = exec C_Delta C_AAM
```

4.4 Recovering kCFA

We skip the recovery of 0CFA in the monadic semantics and head straight to kCFA. The only thing that changes from the monadic point of view is the arbiter of nondeterminism at the bottom of the transformer stack. Rather than `Point` we use the `Nondet` monad as originally introduced in section **??**.

kCFA can be recovered with monad stack

```
type A_M d aam = StateT (E aam) (StateT (S d aam) (StateT (T aam) Nondet))
```

and the `Delta` and `AAM` components described in section 3.7

```
execSymkCFA :: Int -> Call -> SS (A_M Sym_Delta ZCFA_AAM) Call
execSymkCFA k = exec Sym_Delta (kCFA_AAM k)
```

4.5 Optimizations

Until this point there has been no apparent reason–or big payoff–for using the generalized monadic approach. In this section we show how both heap widening and abstract garbage collection can be recovered seamlessly from the monadic abstraction.

4.5.1 Heap Widening

Widening can be recovered by simply propagating the state transformer containing the heap to the very bottom of the stack. The state space

```
[(Call, Env aam, Store d aam, Time aam)]
```

induced by the abstract monad

```
type A_M d aam = StateT (E aam) (StateT (S d aam) (StateT (T aam) Nondet))
```

must turn into

```
[((Call, Env aam, Time aam), Store d aam)]
```

the desired state space for heap widening. We show that this state space is *induced purely by a reordering of the monads* in `A_M`, lifting `Nondet` up one level and `StateT (S d aam)` to the bottom.

```
type A_Widen_M d aam = StateT (E aam) (StateT (T aam) (NondetT (State (S d aam))))
```

This permutation is only valid if the resulting monad adheres to the `Semantics` interface, which we justify next.

To show that `A_Widen_M` implements `Semantics` we need to invent a transformer variant of `Nondet`, which we call `NondetT`. Rather than a pure list of values, `NondetT` carries a list of values inside a monadic action.

```
newtype NondetT m a = NondetT { runNondetT :: m [a] }
```

Aside on List Monad Transformers *It is folklore that the naive implementation of* `ListT m a ≡ m [a]` *is not a monad. Variants of* `ListT m a` *have been proposed which are more akin to monadic streams of* `a` *values in the monad* `m`*. We are able to interpret* `m [a]` *as both a monad and monad plus given extra conditions on the underlying monad, and is to our knowledge the first instance where* `m [a]` *is given a valid monadic instance.*

The `Monad` instance for `NondetT` requires the underlying monad to be a `JoinLattice` *functor*.

```
instance (JoinLatticeF m) => Monad (NonDetT m) where
  return x = NonDetT $ return [x]
  xM >>= k = case joinLatticeF :: JoinLatticeW (m [b]) of
    JoinLatticeW -> do
      xs <- runNondetT xM
      listJoin $ map (runNonDetT . k) xs
```

`NondetT` also propagates the `MonadPlus` interface

```
instance (JoinLatticeF m) => MonadPlus (NonDetT m) where
  mzero = NonDetT mzero
  mplus xM yM = case joinLatticeF :: JoinLatticeW (m [a]) of
    JoinLatticeW -> runNondetT xM \/ runNonDetT yM
```

the `MonadState` interfaces

```
instance (MonadState s m) => MonadState s (NondetT m) where
  get = NondetT $ do
    s <- get
    return [s]
  put s = NondetT $ do
    put s
    return [()]
```

and implements `MonadStep`.

```
instance (Pointed m) => Pointed (NondetT m) where
  point x = NondetT $ point [x]
instance (MonadStep m) => MonadStep (NondetT m) where
  type SS (NondetT m) = NondetT m
  step (f :: a -> NondetT m b) xM =
    NondetT $
    step (\ xs -> runNondetT $ NondetT (return xs) >>= f) $
    runNondetT xM
```

Given the permuted state space, heap widening is recovered merely by swapping `A_M` with `A_Widen_M`.

```
execSymkCFAWiden :: Int -> Call -> SS (A_Widen_M Sym_Delta ZCFA_AAM) Call
execSymkCFAWiden k = exec Sym_Delta (kCFA_AAM k)
```

4.5.2 Abstract Garbage Collection

With the semantics in monadic style, abstract garbage collection can be implemented once–generic to the underlying monad–integrating seamlessly into both heap-cloning and heap-widening semantics.

To implement garbage collection we write a monad-generic function that crawls the expression, environment and store, and replaces the store with a copy containing only reachable addresses. This implementation captures the generic garbage collection algorithm, and scales to both heap-cloning and heap-widening stacks. In fact, it's the exact algorithm used to garbage collect concrete semantics, and behaves as concrete garbage collection when instantiated to concrete semantics parameters.

Detecting free variables is a simple recursive function on the `Call` expression in the state space.

```
free :: Call -> Env aam -> Set (Addr aam)
```

Collecting the set of all reachable addresses demands a fixpoint computation, iterating from the free variables as roots.

```
reachable :: Store d aam -> Set (Addr aam) -> Set (Addr aam)
```

Abstract garbage collection replaces the existing heap with the abstract garbage collected heap.

```
gc :: (Semantics d aam m) => Call -> m ()
```

5 Correctness

5.1 Abstract Semantics

5.2 0CFA

5.3 kCFA

5.4 Widening

5.5 GarbageCollection

5.6 LatticeOfAnalyses

6 Conclusion

References

[1] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, volume 45 of *ICFP '10*, pages 51–62, New York, NY, USA, September 2010. ACM.