A Modular Monadic Framework for Abstract Interpretation (Draft)

David Darais

April 24, 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.

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
```

Evaluating a `Call` expresssions might not terminate, and therefore we cannot compute a denotation for `Call`. 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 with `op` and are striaghtforward.

```
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 with `atom`.

```
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 with `call`.

```
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 by a finite abstract state space. 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 also 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.

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 aam` with associated types `Addr aam` and `Time aam`.

```
class AAM aam where
  type Addr (aam :: *) :: *
  type Time (aam :: *) :: *
  tzero :: aam -> Time aam
  tick :: aam -> 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.)

Following the AAM methodology, we realize that in a world with finite addresses but potentially infinite executions, there is a chance we will need to reuse an address a potentially unbounded number of times. 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.

```
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.

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 delta` and associated type `Val delta`.

```
class Delta delta where
  type Val delta :: * -> *
  lit :: delta -> Lit -> Val delta aam
  clo :: delta -> [Name] -> Call -> Env aam -> Val delta aam
  op :: delta -> Op -> Val delta aam -> Maybe (Val delta aam)
  elimBool :: delta -> Val delta aam -> Set Bool
  elimClo :: delta -> Val delta aam -> Set ([Name], Call, Env aam)
type Env addr = Map Name addr
type Store addr = Map addr (Set (Val addr))
type StateSpace addr = Set (Call, Env addr, Store addr)
```

(Final) Delta Abstraction

The reason for replacing the algebraic data type `Val` with type class `Delta delta`

is that different analyses may want to make different choices in their value representations. 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 user of the semantics as part of the `Delta` interface. `atom` must follow another level of indirection in variable lookups, and must account for nondeterminism.

```
atom :: delta -> Env aam -> Store delta aam -> Atom -> Set (Val delta aam)
atom delta _ _ (LitA l) = setSingleton $ lit delta l
atom _     e s (Var x) = case mapLookup x e of
  Nothing -> Nothing
  Just l -> mapLookup l s
atom delta e s (Prim o a) = case atom delta e s a of
  Nothing -> Nothing
  Just vS -> setPartialMap (op delta o) vS
atom delta e _ (Lam xs c) = setSingleton $ clo delta xs c e
```

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

```
call :: delta -> aam -> Time aam -> Env aam -> Store delta aam -> Call -> Set (Call, Env aam
call delta aam t e s (If a tc fc) =
  eachInSet (atom delta e s a) $ \ b ->
    setSingleton $ (if b then tc else fc, e, s)
call delta aam t e s (App fa xas) =
  eachInSet (atom delta e s fa) $ \ v ->
    eachInSet (elimClo delta 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 allocating addresses, in case the address is already in use.

```
bindMany :: aam -> Time aam -> [Name] -> [Atom] -> Env aam -> Store delta aam -> Maybe (Env
bindMany _   _ [] [] e s = Just (e, s)
bindMany aam t (x:xs) (xa:xas) e s = case bindMany aam xs xas e 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 -> aam -> Set (Call, Env aam, Store delta aam, Time aam) -> Set (Call, Env aam
step delta aam ss = eachInSet ss $ \ (c,e,s,t) ->
  eachInSet (call delta aam t c e s) $ \ (c',e',s') ->
    setSingleton (c',e',s',tick aam c' t)

exec :: delta -> aam -> Call -> Set (Call, Env aam, Store delta aam, Time aam)
exec delta aam c0 = iter (collect (step delta aam)) $ setSingleton (c0, mapEmpty, mapEmpty,
```

The collecting semantics keep track of all previously seen states. This means the state being iterated is purely monotonic, and therefore the iteration must terminate if given a finite instantiation of `delta` and `aam`.

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 axis of parameterization, we first show how to recover a concrete interpreter.

Out semantics has two axis of parameterization: abstract address, value space and delta functions. For abstract address we pick the integers, and for delta function

[1] David Van Horn and Matthew Might. Abstracting abstract machines. *SIGPLAN Not.*, 45(9):51–62, September 2010.