CS 7480 – Program Analysis Seminar
November 10, 2017

Ming-Ho Yee
mhyee@ccs.neu.edu

# Push-Down Automata for Higher Order Flow Analysis

## Motivation

(10 min, running 00:00)

- Warmup, k-CFA example from Dimitrios Vardoulakis's dissertation:

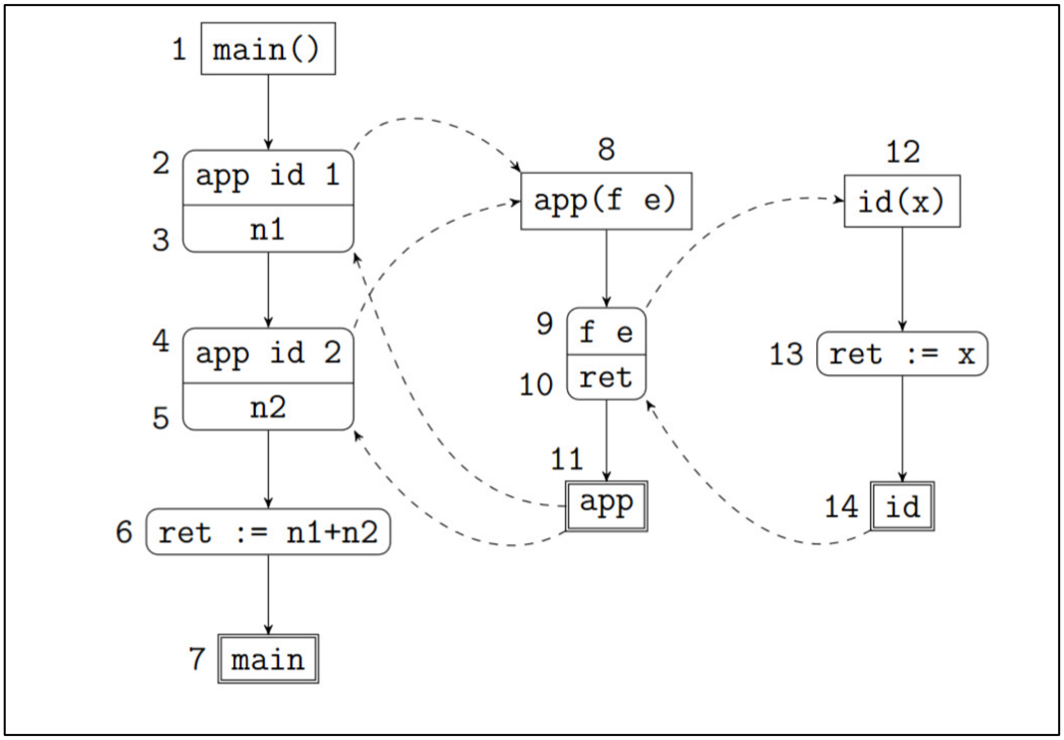
(left board, behind, draw beforehand)

```
(def app (λ (f e) (f e)))
(def id  (λ (x) x))

(let ((n1 (app id 1))
      (n2 (app id 2)))
  (+ n1 n2))
```

- If we step through this program by hand
    - n1 is 1, n2 is 2, result is 3
- 0CFA control-flow graph

(right board, in front, draw beforehand)



- 0CFA allocates a single contour
    - When we call (app id 1)
        - f bound to id, e bound to 1
        - When we call (f e), x bound to 1
        - Return, so n1, n2 bound to 1
    - Then we call (app id 2)
        - f bound to id, e bound to 2
        - When we call (f e), x bound to 2
        - Return, so n1, n2 bound to 2

(left board)

$ENV_0$ =
f -> {id}, e -> {1, 2}, x -> {1,2}
ret_id -> {1,2}, ret_app -> {1,2}
n1 -> {1, 2}, n2 -> {1, 2}
result -> {2, 3, 4}

## Call/Return Mismatch

- Let's try to increase precision with 1CFA
    - 1CFA allocates a contour for the last call site
    - We can differentiate the two calls to app, from sites 2 and 4
    - But id is called from site 9, so we cannot distinguish the contexts

(left board)

$ENV_2$ =
f -> {id}, e -> {1}
ret_app -> {1,2}

$ENV_4$ =
f -> {id}, e -> {2}
ret_app -> {1,2}

$ENV_9$ =
x -> {1, 2}
ret_id -> {1,2}

$ENV_0$ =
n1 -> {1, 2}, n2 -> {1, 2}
result -> {2, 3, 4}

- 1CFA not good enough, so let’s try 2CFA
    - 2CFA allocates a contour for the last 2 call sites
    - Now we can differentiate the calls to id: 9,2 and 9,4
    - Environment maps:

(left board)

$\mathrm{ENV}_2$ =
f -> {id}, e -> {1}
ret_app -> {1,2}

$\mathrm{ENV}_{9,2}$ =
x -> {1}
ret_id -> {1}

$\mathrm{ENV}_0$ =
n1 -> {1}, n2 -> {2}
result -> {3}

$\mathrm{ENV}_4$ =
f -> {id}, e -> {2}
ret_app -> {1,2}

$\mathrm{ENV}_{9,4}$ =
x -> {2}
ret_id -> {2}

- In this example, 2CFA was good enough
    - But given any k, can construct an adversary by eta-expansion
    - Also, k > 1 is already intractable (worse than exponential time)

- Real problem: mismatched calls and returns
    - Approximate program with finite-state machine
        - Cannot “remember” where a call should return to
    - Use a more powerful abstraction: pushdown automata
        - When calling, push onto stack
        - When returning, check top of stack and pop

## Performance and the Vicious Cycle

- Imprecision can lead to worse performance
- Imprecision means more spurious control-flow paths
- More control-flow paths means more to analyze

# CFA2: a Context-Free Approach to Control-Flow Analysis

(20 min, running 10:00)

**Vardoulakis and Shivers, ESOP 2010**

- CFACFA = CFA2, *not* 2-CFA
- "Context-Free" language

## Concrete Semantics

- Standard recipe for analysis: formalize the concrete semantics
  - Continuation-Passing Style
  - eval-apply interpreter

## Abstract Semantics

- CFA2 is an abstract interpretation of the CPS program

(cover left board)

(right board)

1. **Split environment into stack/heap**

   `(`$\lambda_1$`(x) (`$\lambda_2$`(y) (y (y x))))`

   Stack ref: y Heap ref: x
2. **Use stack for variable binding, return-point info**
3. **Concrete states -> abstract states**

- Reference is a stack ref if it appears at same nesting level as its binder
  - Inner lambda and its reference to x can escape
- Possible to come up with a definition for stack/heap references in CPS
- In general, multiple closures may flow to f
  - And we might choose different values for the different calls
  - But in this case, both references are bound at the same time
  - We update the top frame with the value we chose for y

- Transform concrete states to abstract states
    - Transform environment into a stack
    - Make the environment finite, allow sets of values
    - Update transitions
- Now we have a semantics that accurately describes call/return matching

## Local Semantics

- Abstract state space is infinite due to the stack, so not computable

(right board)

**4. Abstract states → local states**

- Keep top stack frame
- Drop rest of stack

- Map abstract states to local states
    - Functions do not return

## Summarization

(right board)

**5. Summarization**

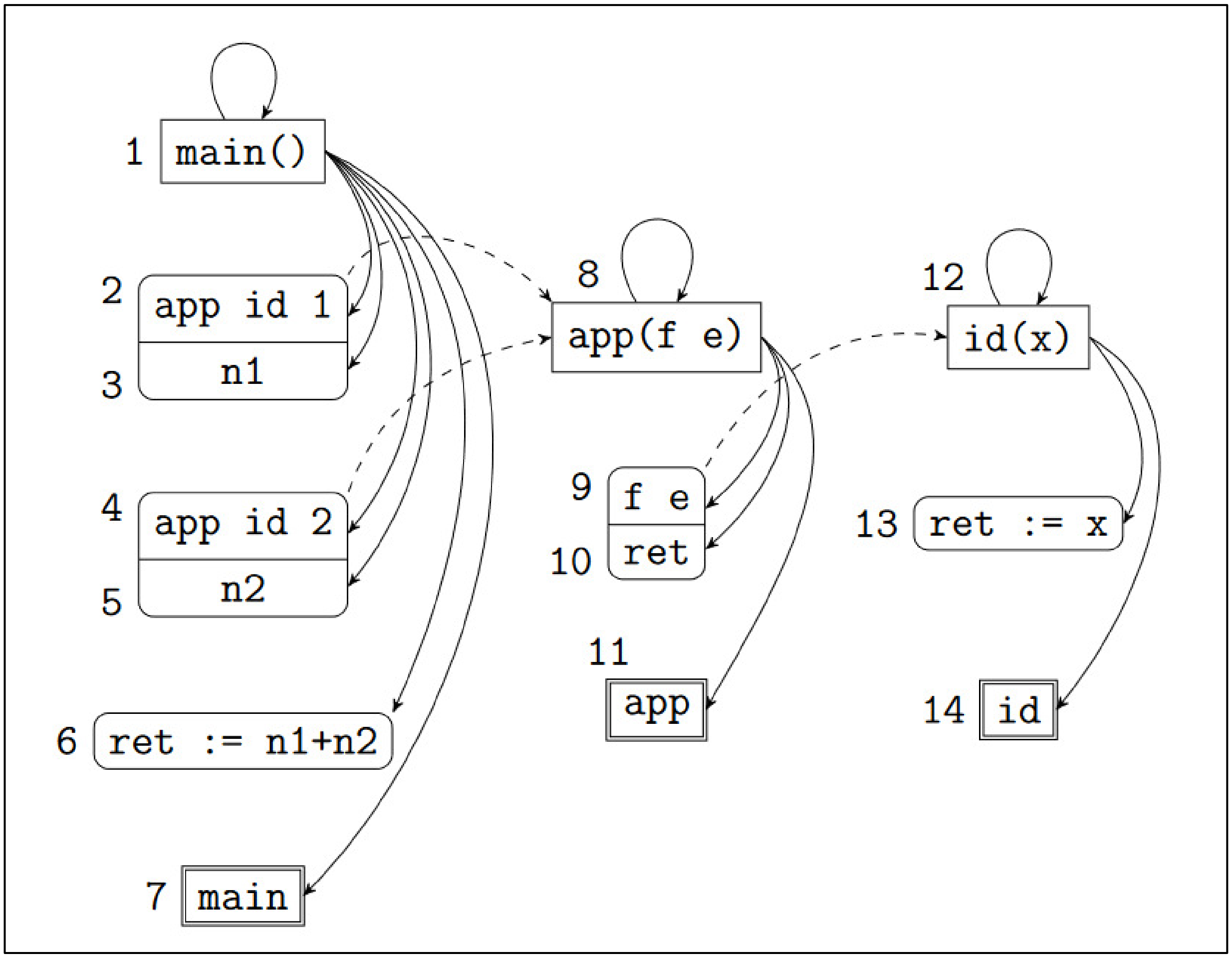
Path edge: entry node -> some node in same procedure

Summary edge: entry node -> exit node

- Summarization is a dynamic programming algorithm
- Graph reachability problem
- Example uses nodes, but algorithm uses states (includes heap and top frame)

Ming-Ho Yee
mhyee@ccs.neu.edu

(reuse left board)



| PathEdge | SummaryEdge | Callers |
|---|---|---|
| <1,1>,<1,2> | | <1,2,8> |
| No summary found | | |
| <8,8>, <8,9> | | <8,9,12> |
| No summary found | | |
| <12,12>,<12,13>,<12,14> | <12,14> | |
| Found Caller<8,9,12>, return to 10 | | |
| <8,10>,<8,11> | <8,11> | |
| Found Caller<1,2,8>, return to 3 | | |
| <1,3>,<1,4> | | <1,4,8> |
| Found Summary<8,11>, return to 5 | | |
| <1,5>,<1,6>,<1,7> | | |

Ming-Ho Yee
mhyee@ccs.neu.edu

## Complexity and Evaluation

- Complexity: worse than exponential
    - Exploring states,
    - Each state has $h \in Heap = Var \rightarrow Pow(Clo)$
    - $Var \in O(n)$
    - $Pow(Clo) \in O(2^n)$
    - $Heap \in O\left(2^{n^2}\right)$
- But seems to be OK in practice

- Evaluation in paper compares 0CFA, 1CFA, and CFA2
    - Precision: CFA2 most precise, then 1CFA, then 0CFA
    - Efficiency: 1CFA worse, 0CFA and CFA2 about the same

# Pushdown Control-Flow Analysis for Free

(30 min, running 30:00)

**Gilray, Lyde, Adams, Might, Van Horn, POPL 2016**

(right board, in front)

- PDCFA (Pushdown Control-Flow Analysis)
  - Complex implementation, $O(f(n)^2)$
- AAC (Abstracting Abstract Control)
  - Simple implementation, $O(n^2 f(n)^2)$
- P4F (this paper)
  - Simple implementation, $O\big(f(n)\big)$

- Other groups were working on the same problem at the same time
  - Based on the AAM approach
  - Culminates in this paper

- Quick review of AAM
- Use A-Normal Form as the intermediate representation
  - Like CPS, avoids nested calls
  - Uses let-bindings for intermediate expressions
  - Order of operations explicit from let-bindings

## Concrete Semantics

(left board, behind)

| | | |
|---|---|---|
| $\varsigma \in \Sigma$ | $= \mathrm{Exp} \times Env \times Store \times Kont$ | [states] |
| $\rho \in Env$ | $= \mathrm{Var} \rightarrow Addr$ | [environments] |
| $\sigma \in Store$ | $= Addr \rightarrow Clo$ | [stores] |
| $clo \in Clo$ | $= \mathrm{Lam} \times Env$ | [closures] |
| $\kappa \in Kont$ | $= Frame^*$ | [stacks] |
| $\phi \in Frame$ | $= \mathrm{Var} \times \mathrm{Exp} \times Env$ | [stack frames] |
| $a \in Addr$ | infinite set | [addresses] |

```
let id = (λ (z) z)
let  x = (id v)
let  y = ...
```

- We always generate a fresh address
- Two kinds of transitions: calls and returns
- Call transition:
  - Push new stack frame so we know how to return
  - Evaluate function and its arguments
  - Bind arguments to formals and update store/env
- Return transition:
  - Pop the stack
  - Restore the old environment
  - Bind result to the variable in the frame
  - Transfer control to expression in the frame

## Abstract Semantics

- Make domains finite so we can compute the analysis
    - Unboundedness: store (due to addresses) and stacks

(right board, in front)

| | | |
|---|---|---|
| $\tilde{\varsigma} \in \tilde{\Sigma}$ | $= \mathrm{Exp} \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore} \times \widetilde{Addr}$ | [states] |
| $\tilde{\rho} \in \widetilde{Env}$ | $= \mathrm{Var} \rightarrow \widetilde{Addr}$ | [environments] |
| $\tilde{\sigma} \in \widetilde{Store}$ | $= \widetilde{Addr} \rightarrow \mathcal{P}(\widetilde{Clo})$ | [val. stores] |
| $\widetilde{clo} \in \widetilde{Clo}$ | $= \mathrm{Lam} \times \widetilde{Env}$ | [closures] |
| $\widetilde{\sigma_\kappa} \in \widetilde{KStore}$ | $= \widetilde{Addr} \rightarrow \mathcal{P}(\widetilde{Kont})$ | [cont. stores] |
| $\tilde{\kappa} \in \widetilde{Kont}$ | $= \widetilde{Frame} \times \widetilde{Addr}$ | [continuations] |
| $\tilde{\phi} \in \widetilde{Frame}$ | $= \mathrm{Var} \times \mathrm{Exp} \times \widetilde{Env}$ | [stack frames] |
| $\tilde{a}, \widetilde{a_\kappa} \in \widetilde{Addr}$ | finite set | [addresses] |

- Value store: map finite set of addresses to set of abstract closures
- Stack: thread through store as linked list, represent continuation with address
    - Continuation is a (top) frame and an address to the next continuation (rest of stack)
    - We can merge frames, and we might have a cycle so it's finite

- How do we pick addresses from a finite set? Describe it using an abstract allocator

(left board)

$\widetilde{alloc}$ : Var × $\tilde{\Sigma} \rightarrow \widetilde{Addr}$

Var – variable we want an address for
$\tilde{\Sigma}$ – current state

E.g.: $\widetilde{alloc}_0(x, \tilde{\varsigma}) = x$

- In 0-CFA, allocator just uses the variable as its address
  - Corresponds to a single global environment
- Also need an abstract continuation allocator

(left board)

$\widetilde{alloc}_\kappa$ : $\tilde{\Sigma}$ × Exp × $\widetilde{Env} \times \widetilde{Store} \rightarrow \widetilde{Addr}$

$\tilde{\Sigma}$ – current state
Exp – target expression
$\widetilde{Env}$ – target environment
$\widetilde{Store}$ – target store

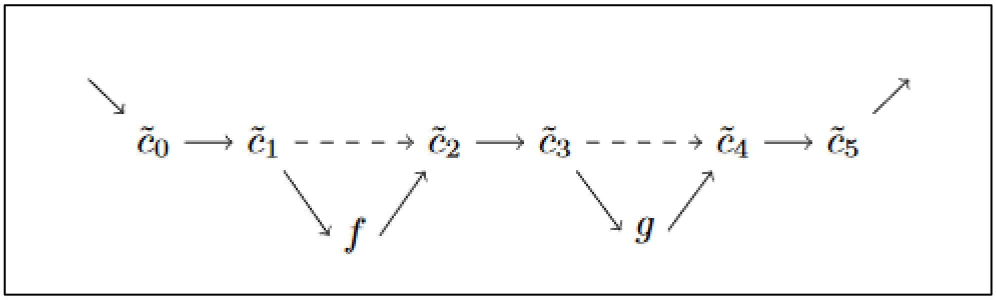
E.g.: $alloc_{\kappa 0}(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}') = e'$

- Don't have $\widetilde{\sigma_\kappa}$ or $\widetilde{a_\kappa}$ because we need the allocator to give us an address
- Example: use the target expression (where function returns to) as abstract address
  - If we used CPS, the 0CFA store allocator would give us this

## Pushdown 4 Free

(left board)

(use $\varsigma$ instead of c)



- We call a procedure, entering at $s_0$ and exiting at $s_5$
  - We enter with some amount of precision
  - E.g. 2 different call sites, so 2 different entry configurations
- We want the continuations to have at least the same amount of precision
  - Continuation allocator should be as precise as the value allocator
- Target environment is determined by its addresses, which are determined by the value allocator

(right board, bottom)

$$alloc_{\kappa\ P4F}(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}') = (e', \widetilde{\rho'})$$

- Abstract address is the target expression and target env
- No matter what value allocator you choose, this continuation allocator will give you a precise address

Ming-Ho Yee
mhyee@ccs.neu.edu

## Example

(move right board to left)
(right board, behind)

```
… ¹(let ([y (id #t)])
    ²(let ([z (id #f)])
      ³…)
```

Initial values:

$\texttt{id -> ((λ (x) }{}^{0}\texttt{x), }\widetilde{\rho_\lambda}\texttt{)}$

$\tilde{\rho}, \widetilde{a_\kappa}$

$\widetilde{alloc}_1(x, (e, \tilde{\rho}, \tilde{\sigma}, \widetilde{\sigma_\kappa}, \widetilde{a_\kappa})) = (x, e)$

- 1-CFA allocator: use the variable and call site as address
- Simple continuation allocator: use target address
- First step: apply id to #t, so we enter the body of id and update our stores
    - Note that in P4F we also use the target environment

| $\tilde{\sigma}$ - store | $\kappa$ – cont. | $\widetilde{\sigma_\kappa}$ – cont. store (imprecise) | $\widetilde{\sigma_{\kappa\ P4F}}$ – cont. store (precise) |
|---|---|---|---|
| $(x, e_1) \mapsto \{\#t\}$ | $\kappa_1 = ((y, e_2, \tilde{\rho}), \widetilde{a_\kappa})$ | $e_0 \mapsto \{\kappa_1\}$ | $(e_0, \widetilde{\rho_\lambda}[x \mapsto (x, e_1)]) \mapsto \{\kappa_1\}$ |

- Now we return from $e_0$ to $e_2$ and bind y to the result and update stores

| | | | |
|---|---|---|---|
| $(y, e_0) \mapsto \{\#t\}$ | | | |

- Second call, but this time we apply id to #f
    - Imprecise return address is $e_0$, precise is ($e_0$, \rho …)

| | | | |
|---|---|---|---|
| $(x, e_2) \mapsto \{\#f\}$ | $\kappa_2 = ((z, e_3, \tilde{\rho}[y \mapsto (y, e_0)]), \widetilde{a_\kappa})$ | $e_0 \mapsto \{\kappa_1, \kappa_2\}$ | $(e_0, \widetilde{\rho_\lambda}[x \mapsto (x, e_1)]) \mapsto \{\kappa_1\}$<br>$(e_0, \widetilde{\rho_\lambda}[x \mapsto (x, e_2)]) \mapsto \{\kappa_2\}$ |

- Return from $e_0$

| | | | |
|---|---|---|---|
| $(z, e_0) \mapsto \{\#f\}$ | | $e_0 \mapsto \{\kappa_1, \kappa_2\}$ | $(e_0, \widetilde{\rho_\lambda}[x \mapsto (x, e_1)]) \mapsto \{\kappa_1\}$<br>$(e_0, \widetilde{\rho_\lambda}[x \mapsto (x, e_2)]) \mapsto \{\kappa_2\}$ |

- Both precise and imprecise analysis correctly bind z
- But imprecise analysis sees $e_0$ bound to two continuations, so we also return to $e_2$
    - (y, $e_0$) gets bound to #t and #f

## Abstracting Definitional Interpreters

(30 min, running 1:00:00)

**Darais, Labich, Nguyễn, Van Horn, ICFP 2017**

- Now for something that seems unrelated, but based on AAM
- Idea: instead of applying abstract interpretation to an abstract machine, apply abstract interpretation to definitional interpreter
    - High level, reusable, extensible
    - Inherits the "pushdown control flow" property from the metalanguage
- "Definitional interpreters" and "inheritance" come from Reynolds 1972: Definitional Papers for Higher-order Programming Languages

(left board)

```
(def (eval exp env)
  (match exp
    [(vbl v)     (lookup env v)]
    [(app e₀ e₁) ((eval e₀ env) (eval e₁ env))]
    [(lam x e)   (λ (v) (eval e (extend env x v)))]))
```

- Defined language: untyped lambda calculus
- Defining language (or metalanguage): Racket-like language

- If the metalanguage is call-by-value, so is the defined language
- If the metalanguage is call-by-name, so is the defined language
- Defined language "inherits" evaluation strategy from metalanguage

- Interpreter uses monads, but why?
- Try writing an arithmetic evaluator that handles errors
    - Use the Maybe monad, which is called Option or Optional

(right board)

Maybe ::= Just n | Nothing

```
(define (add mx my)
  (match x
    [(Nothing) (Nothing)]
    [(Just x)  (match my
                 [(Nothing) (Nothing)]
                 [(Just y)  (Just (+ x y))])]))
```

- A lot of "noise" that obscures the actual important computation
- Library that provided Maybe also provides operations for chaining values together

(right board)

```
(define (return v)
  (Just v))

(define (bind mv f)
  (match mv
    [(Nothing) (Nothing)]
    [(Just v)  (f v)]))
```

- bind takes a Maybe value and a function
    - If the value is Nothing, "short circuit" and return Nothing
    - Otherwise apply function to the unwrapped value
    - Note that f must return a Maybe value of its own
- Return takes a value and wraps it up as a Maybe value
- There are also 3 "monad laws" that return and bind must obey
- Now let's rewrite the add function

(left board)

```
(define (add mx my)
  (bind mx (λ (x)
  (bind my (λ (y)
  (return (+ x y)))))))
```

```
(define (add mx my)
  (do x <- mx
      y <- my
      (return (+ x y)))
```

- We have a Maybe value mx, which we “unwrap” and bind to x
  - We unwrap my and bind to y
  - Then we can add x+y and “return”, which wraps the result
- If mx or my are Nothing, then we skip all the computation and return Nothing
- Problem: still kind of ugly, so we use do-notation
  - Fun fact: now looks like imperative programming

- Can switch to a “Nondeterminism” monad (with its implementation of ‘bind’ and ‘return’) that represents set of values
  - Now ‘add’ can add sets of values and return a set of all possible sums
- Phil Wadler showed how to write an interpreter, where you could “plug in” different monads to get different effects
- Follow-up work showed how you could use “monad transformers” to compose monads and get different combinations of effects

- Now let’s go back to the interpreter (simplified)
  - State monad for store, Reader monad for environment

(right board)

```
(define ((ev ev’) e)   ; env=var->addr   store=addr->val
  (match e
    [(num n)       (return n)]
    [(vbl x)       (do ρ <- ask-env
                       (find (ρ x)))]
    [(lam x e₀)    (do ρ <- ask-env
                       (return (cons (lam x e₀) ρ)))]
    [(app e₀ e₁)   (do (cons (lam x e₂) ρ) <- (ev’ e₀)
                       v₁ <- (ev’ e₁)
                       a  <- (alloc x)
                       (ext a v₁)
                       (local-env (ρ x a) (ev’ e₂)))]))
```

- Like AAM, we have an environment (variable -> address) and a store (address -> value)
- Some interesting points
    - Written in “open recursive style” where it calls the argument ev’
        - Need to apply Y combinator to get recursion
        - Purpose: intercept recursive calls
    - Underlined parts are incomplete, subject to the component we “plug in”
        - Bind/return are for the underlying monad
        - Environment: ask-env to retrieve, local-env to restore
        - Store: find to dereference, ext to update, alloc to allocate

## Concrete Interpreter

(left board)

```
(define (alloc x) (do σ <- get-store
                      (return (size σ))))
```

- Need implementations for all the underlined functions
- Concrete ‘alloc’ might be implemented like this
  - Returns the size of the store
  - So every time we add an address to the store, we’ll get a fresh address

## Abstract Interpreter

(left board)

```
(define (alloc x) (return x))
```

- Now to abstract our interpreter
- Abstract allocator returns an address from a finite set
  - E.g. 0CFA uses the name of the variable as its address
- If the interpreter handled values, we would need to abstract values
  - E.g. concrete numbers represented by their sign
  - If we have branching, need to use nondeterminism to take both branches
- Store: map addresses to set of values, update is join, dereference is nondeterministic

## Termination

(left board)

AAM – transitions over finite state space

ADI – caching fixed-point algorithm

- The interpreter and abstraction were easier
- But guaranteeing termination is the trickiest bit; details in paper
    - Cache visited configurations
    - Cache results
    - Compute least fixed point of the cache
- Now we have a terminating abstract interpreter
- Skipped one step from AAM: no store-allocated continuations
    - No continuations
- Stack is implicit, modeled by the metalanguage (Racket) and not the interpreter
    - Racket is precise and does call/return matching
    - Therefore, abstract interpreter is also precise