

Abstracting Definitional Interpreters

David Darais
University of Maryland

Nicholas Labich
University of Maryland

Phúc C. Nguyễn
University of Maryland

David Van Horn
University of Maryland

Does my program cause a runtime error?

Does my program allocate too much?

Does my program sanitize all untrusted inputs?

Is this proof object computationally relevant?



**My PL Doesn't Have
a Program Analyzer**



**Should I Write My Own
Program Analyzer?**



Writing Your Own Program Analyzer is Easy

If you know how to write an interpreter

Abstracting Definitional Interpreters

Interpreter \Rightarrow Analyzer

Sound Terminating Precise Extensible

Context:

Abstracting Abstract Machines (AAM): [ICFP '10]

Sound + Terminating + Easy

Based on low-level *Abstract Machines*

Context:

Abstracting Abstract Machines (AAM): [ICFP '10]

Sound + Terminating + Easy

Based on low-level *Abstract Machines*

This Paper:

Abstracting Definitional Interpreters (ADI): [ICFP '17]

Sound + Terminating + *Extra Precision* + *Even Easier*

Based on high-level *Definitional Interpreters*

Inheriting Precision

Reynolds - Inheriting properties from defining language
[1972]

This work - Inherit *analysis precision* from the metalanguage

Result - *pushdown analysis*

Many papers on pushdown precision; we get it for free

Key Challenges

Soundness:

AAM: A single (parameterized) *machine* recovers both concrete and abstract semantics

ADI: A single (parameterized) *interpreter* recovers both concrete and abstract semantics

Key Challenges

Soundness:

AAM: A single (parameterized) *machine* recovers both concrete and abstract semantics

ADI: A single (parameterized) *interpreter* recovers both concrete and abstract semantics

Termination:

AAM: Iterating a transition system with finite state space

ADI: Caching fixpoint algorithm for unfixed interpreters

Concrete Interpreter

Partial Abstract Interpreter

Total Abstract Interpreter

Concrete Interpreter

1. Store-allocation style for argument binding
2. Monadic environment and state
3. Parameters for primitive operators and allocation
4. “Unfixed” style

```
; m is monad  
; m is monad-reader[env]  
; m is monad-state[store]  
; ev : exp → m(val)
```

```
env := var ↦ addr  
store := addr ↦ val
```

```
; m is monad
; m is monad-reader[env]          env = var ↦ addr
; m is monad-state[store]        store = addr ↦ val
; ev : exp → m(val)
(define (ev e)
  (match e
    [(num n)      (return n)]
    [(vbl x)      (do ρ ← ask-env
                    (find (lookup x ρ)))]
```



```

; m is monad
; m is monad-reader[env]          env = var ↦ addr
; m is monad-state[store]        store = addr ↦ val
; ev : exp → m(val)
(define (ev e)
  (match e
    [(num n)      (return n)]
    [(vbl x)      (do ρ ← ask-env
                     (find (lookup x ρ)))]
    [(if0 e1 e2 e3) (do v ← (ev e1)
                       z? ← (zero? v)
                       (ev (if z? e2 e3)))]
    [(op2 0 e1 e2) (do v1 ← (ev e1)
                      v2 ← (ev e2)
                      (δ 0 v1 v2))])

```

```

; m is monad
; m is monad-reader[env]      env = var ↦ addr
; m is monad-state[store]    store = addr ↦ val
; ev : exp → m(val)
(define (ev e)
  (match e
    [(num n)      (return n)]
    [(vbl x)      (do ρ ← ask-env
                     (find (lookup x ρ)))]
    [(if0 e1 e2 e3) (do v ← (ev e1)
                       z? ← (zero? v)
                       (ev (if z? e2 e3)))]
    [(op2 o e1 e2) (do v1 ← (ev e1)
                       v2 ← (ev e2)
                       (δ o v1 v2))]
    [(lam x e)     (do ρ ← ask-env
                     (return (cons (lam x e) ρ)))]
    [(app e1 e2)   (do (cons (lam x e') ρ') ← (ev e1)
                       v2 ← (ev e2)
                       a ← (alloc x)
                       (ext a v2)
                       (local-env (update x a ρ') (ev e')))]))

```

```

; m is monad
; m is monad-reader[env]      env = var ↦ addr
; m is monad-state[store]    store = addr ↦ val
; ev : (exp → m(val)) → exp → m(val)
(define ((ev ev') e)
  (match e
    [(num n)      (return n)]
    [(vbl x)      (do ρ ← ask-env
                      (find (lookup x ρ)))]
    [(if0 e1 e2 e3) (do v ← (ev' e1)
                        z? ← (zero? v)
                        (ev' (if z? e2 e3)))]
    [(op2 o e1 e2) (do v1 ← (ev' e1)
                       v2 ← (ev' e2)
                       (δ o v1 v2))]
    [(lam x e)     (do ρ ← ask-env
                      (return (cons (lam x e) ρ)))]
    [(app e1 e2)   (do (cons (lam x e') ρ') ← (ev' e1)
                       v2 ← (ev' e2)
                       a ← (alloc x)
                       (ext a v2)
                       (local-env (update x a ρ') (ev' e')))]))

```

Running The Interpreter

```
; Y : ((a → m(b)) → a → m(b)) → a → m(b)
(define ((Y f) x)
  ((f (Y f)) x))

; eval : exp → val × store
(use-monad (ReaderT env (StateT store ID)))
(define (eval e)
  (mrun ((Y ev) e)))
```

Running The Interpreter

```
; Y : ((a → m(b)) → a → m(b)) → a → m(b)
(define ((Y f) x)
  ((f (Y f)) x))

; eval : exp → val × store
(use-monad (ReaderT env (StateT store ID)))
(define (eval e)
  (mrun ((Y ev) e)))

> ((λ (x) (λ (y) x)) 4)
'(((λ (y) x) . ((x . 0))) . ((0 . 4)))
```

Interpreter Extensions

Intercept recursive calls in the interpreter

Change monad parameters

Change primitive operators and allocation

E.G., A Tracing Analysis

```
; m is monad
; m is monad-reader[env]
; m is monad-state[store]
; m is monad-writer[config]
; ev-trace : ((exp → m(val)) → exp → m(val))
             → (exp → m(val)) → exp → m(val)
(define (((ev-trace ev) ev') e)
  (do  $\rho$  ← ask-env
       $\sigma$  ← get-store
      (tell (list e  $\rho$   $\sigma$ ))
      ((ev ev') e)))
```

Running the Analysis

```
; eval : exp → (val × store) × list(config)
(use-monad (ReaderT env (WriterT list (StateT store ID))))
(define (eval e)
  (mrrun ((Y (ev-trace ev)) e)))
```


Running the Analysis

```
; eval : exp → (val × store) × list(config)
(use-monad (ReaderT env (WriterT list (StateT store ID))))
(define (eval e)
  (mrun ((Y (ev-trace ev)) e)))
```

```
> (* (+ 3 4) 9)
' ((63 . ())
  (( (* (+ 3 4) 9) () ()))
  (( + 3 4) () ()))
(3 () ())
(4 () ())
(9 () ()))
```

Concrete Interpreter

Partial Abstract Interpreter

Total Abstract Interpreter

Partial Abstract Interpreter

1. Abstracting Primitive Operations
2. Abstracting Allocation

The Game: "Abstract" = finite

Abstracting Numbers

```
; m is monad-failure
; m is monad-nondeterminism
; num :=  $\mathbb{Z} \cup \{N\}$ 

;  $\delta$  : op num num  $\rightarrow$  m(num)
(define (delta o n1 n2)
  (match o
    ['+ (return 'N)]
    ['/ (do z?  $\leftarrow$  (zero? n2)
            (if z? fail (return 'N)))]))
```

Abstracting Numbers

```
; m is monad-failure
; m is monad-nondeterminism
; num :=  $\mathbb{Z} \cup \{N\}$ 

;  $\delta : \text{op num num} \rightarrow \text{m(num)}$ 
(define (delta o n1 n2)
  (match o
    ['+ (return 'N)]
    ['/ (do z? ← (zero? n2)
            (if z? fail (return 'N)))]))

; zero? : num → m(bool)
(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_ (return (= v 0))]))
```

Abstracting Addresses

```
; alloc : var → m(addr)
(define (alloc x)
  (return x))
```

Abstracting Addresses

```
; alloc : var → m(addr)
(define (alloc x)
  (return x))

; ext : addr × val → m(unit)
(define (ext a v)
  (do  $\sigma \leftarrow$  get-store
      (put-store (union  $\sigma$  (dict a (set v))))))
```

Running the Analysis

```
; eval : exp →  $\wp$ (option(val) × store)
(use-monad (ReaderT env (FailT (StateT store (NondetT ID))))
(define (eval e)
  (mrun ((Y ev) e)))
```


Running the Analysis

```
; eval : exp →  $\wp$ (option(val) × store)
(use-monad (ReaderT env (FailT (StateT store (NondetT ID))))
(define (eval e)
  (mrun ((Y ev) e)))

> (let ((f (λ (x) x)))
  (f 1)
  (f 2))
' (set 1 2)
```

Running the Analysis

```
; eval : exp →  $\wp$ (option(val) × store)
(use-monad (ReaderT env (FailT (StateT store (NondetT ID))))
(define (eval e)
  (mrun ((Y ev) e)))

> (let ((f (λ (x) x)))
  (f 1)
  (f 2))
' (set 1 2)

> (letrec ((loop (λ (x) (loop x))))
  (loop 1))
TIMEOUT
```

Concrete Interpreter

Partial Abstract Interpreter

Total Abstract Interpreter

`[(loop 1)]`



`[(loop 1)]`



Total Abstract Interpreters

1. Remember visited configurations

`[(loop 1)]`



`[(loop 1)]`

I've already
seen that
config...

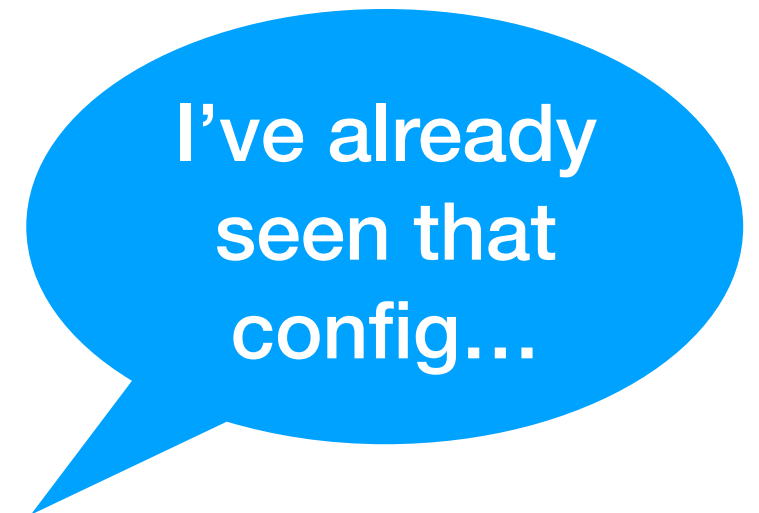
$[(loop\ 1)]$



$[(loop\ 1)]$



\emptyset



I've already
seen that
config...

Total Abstract Interpreters

1. Remember visited configurations

(Sufficient for *termination*)

(Unsound for *abstraction*)


```
[(fact 'N)]
```



```
[(if (zero? 'N)  
1  
(* 'N (fact (- 'N 1))))]
```

```
[[ (fact 'N) ]]
```



```
[[ (if (zero? 'N)  
1  
(* 'N (fact (- 'N 1)))) ]]
```



```
1
```

```
[[ (* 'N (fact (- 'N 1)) ) ]]
```

[[(fact 'N)]]



[[(if (zero? 'N)
1
(* 'N (fact (- 'N 1))))]]



1

[[(* 'N (fact (- 'N 1)))]]



[[(* 'N (fact 'N))]]



'N × [[(fact 'N)]]

[[(fact 'N)]]



[[(if (zero? 'N)
1
(* 'N (fact (- 'N 1))))]]



1 [[(* 'N (fact (- 'N 1)))]]



[[(* 'N (fact 'N))]]



← 'N × [[(fact 'N)]]

I've already seen that config...

$\llbracket (\text{fact } 'N) \rrbracket = \{1\}$



$\llbracket (\text{if } (\text{zero? } 'N)$
1
 $\quad (* 'N (\text{fact } (- 'N 1))) \rrbracket$



1



$\llbracket (* 'N (\text{fact } (- 'N 1))) \rrbracket$



$\llbracket (* 'N (\text{fact } 'N)) \rrbracket$



$'N \times \llbracket (\text{fact } 'N) \rrbracket$

I've already
seen that
config...

$\llbracket (\text{fact } 'N) \rrbracket = \{1\}$ ~~X~~



$\llbracket (\text{if } (\text{zero? } 'N)$
1
 $\quad (* 'N (\text{fact } (- 'N 1)))) \rrbracket$



1

$\llbracket (* 'N (\text{fact } (- 'N 1))) \rrbracket$



$\llbracket (* 'N (\text{fact } 'N)) \rrbracket$



← $'N \times \llbracket (\text{fact } 'N) \rrbracket$

I've already
seen that
config...

Total Abstract Interpreters

1. Remember visited configurations
2. Bottom out to a “cached” result

[[(fact 'N)]]



[[(if (zero? 'N)

1

(* 'N (fact (- 'N 1))))]]



1

[[(* 'N (fact (- 'N 1)))]]



[[(* 'N (fact 'N))]]



'N × [[(fact 'N)]]

`[(fact 'N)]`



`[(if (zero? 'N)
1
(* 'N (fact (- 'N 1))))]`



`1`



`[(* 'N (fact (- 'N 1)))]`



`[(* 'N (fact 'N))]`



`'N × $[(fact 'N)]`

$$\llbracket (\text{fact } 'N) \rrbracket = \{1\} \cup \{ 'N \times \$\llbracket (\text{fact } 'N) \rrbracket \}$$



$\llbracket (\text{if } (\text{zero? } 'N)$
1
 $\quad (* 'N (\text{fact } (- 'N 1))) \rrbracket$



1



$\llbracket (* 'N (\text{fact } (- 'N 1))) \rrbracket$



$\llbracket (* 'N (\text{fact } 'N)) \rrbracket$



$'N \times \$\llbracket (\text{fact } 'N) \rrbracket$

$$[(\text{fact } 'N)] = \{1\} \cup \{ 'N \times \$[(\text{fact } 'N)] \}$$

Q: How to compute $\$[(\text{fact } 'N)]$?

$$\llbracket (\text{fact } 'N) \rrbracket = \{1\} \cup \{ 'N \times \$\llbracket (\text{fact } 'N) \rrbracket \}$$

Q: How to compute $\$\llbracket (\text{fact } 'N) \rrbracket$?

$$\$\llbracket (\text{fact } 'N) \rrbracket \approx \llbracket (\text{fact } 'N) \rrbracket$$

$$\llbracket (\text{fact } 'N) \rrbracket = \{1\} \cup \{ 'N \times \$\llbracket (\text{fact } 'N) \rrbracket \}$$

Q: How to compute $\$\llbracket (\text{fact } 'N) \rrbracket$?

$$\$\llbracket (\text{fact } 'N) \rrbracket \approx \llbracket (\text{fact } 'N) \rrbracket$$

A: Compute least-fixpoint of equations

```
(define (eval e)
  (mrun ((fix-cache (Y (ev-cache ev))) e)))
```



**Intercepts recursion
to call the cache**

```
(define (eval e)  
  (mrun ((fix-cache (Y (ev-cache ev))) e)))
```



**Computes the
least-fixpoint**

```
(define (eval e)
  (mrrun ((fix-cache (Y (ev-cache ev))) e)))

> (letrec ((loop (λ (x) (loop x))))
  (loop 1))
(set)
```



```
(define (eval e)
  (mrrun ((fix-cache (Y (ev-cache ev))) e)))

> (letrec ((loop (λ (x) (loop x))))
  (loop 1))
(set)

> (letrec ((fact (λ (x)
  >           (if0 x
  >             1
  >             (* x (fact (- x 1))))))))
> (fact 6))
(set 'N)
```

Total Abstract Interpreters

1. Remember visited configurations
2. Bottom out to a “cached” result
3. Compute least-fixpoint of the cache

(See full caching algorithm in the paper)

Extra Precision

We've actually recovered *pushdown* OCFA

There is no approximation for stack frames

Call/return semantics is implemented by the *metalanguage* (Racket)

Precise call/return semantics = pushdown precision

What Else is in the Paper?

- Pushdown analysis
- Global store-widening
- A more precise arithmetic abstraction
- (Sound) Symbolic execution
- Abstract garbage collection
- Proof of soundness via big-step reachability semantics (supp. material)



Go and Write Your Own Program Analyzer

It's just a slightly fancy interpreter

Abstracting Definitional Interpreters

Interpreter \Rightarrow Analyzer

Sound Terminating Precise Extensible