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 => 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]          env ≔ var ↦ addr
; m is monad-state[store]         store ≔ addr ↦ val
; ev : exp → m(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 o e₁ e₂)  (do v₁ ← (ev e₁)
                         v₂ ← (ev e₂)
                         (δ o 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 ρ ← ask-env
      σ ← get-store
      (tell (list e ρ σ))
      ((ev 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)))
```

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 ≔ ℤ ⊎ {'N}

; δ : op num num → m(num)
(define (δ o n₁ n₂)
  (match o
    ['+ (return 'N)]
    ['/ (do z? ← (zero? n₂)
          (if z? fail (return 'N)))]))
```

Abstracting Numbers

```
; m is monad-failure
; m is monad-nondeterminism
; num ≔ ℤ ⊎ {'N}

; δ : op num num → m(num)
(define (δ 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 σ ← get-store
      (put-store (union σ (dict a (set v))))))
```

Running the Analysis

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

Running the Analysis

```
; eval : exp → ℘(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 → ℘(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

$$[\![\texttt{(loop 1)}]\!]$$

$$\Downarrow$$

$$[\![\texttt{(loop 1)}]\!]$$

$$\Downarrow$$

...

Total Abstract Interpreters

1. Remember visited configurations

$$[\![\texttt{(loop 1)}]\!]$$

$$\Downarrow$$

$$[\![\texttt{(loop 1)}]\!]$$

I've already seen that config...

⟦`(loop 1)`⟧

$\Downarrow$

⟦`(loop 1)`⟧

$\Downarrow$

$\varnothing$

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…

⟦(fact 'N)⟧ = {1}

⇓

```
⟦(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…

```
⟦(fact 'N)⟧  = {1}  ✗

        ⇓

⟦(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…

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)]
```

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

$$\Downarrow$$

```
⟦(if (zero? 'N)
     1
     (* 'N (fact (- 'N 1))))⟧
```

$$\swarrow \qquad\qquad \searrow$$

```
1        ⟦(* 'N (fact (- 'N 1)))⟧
```

$$\Downarrow$$

```
⟦(* 'N (fact 'N))⟧
```

$$\Downarrow$$

```
'N × $[(fact 'N)]
```

$⟦(\texttt{fact 'N})⟧ = \{1\} \cup \{\texttt{'N} \times \$[(\texttt{fact 'N})]\}$

Q: How to compute `$[(fact 'N)]`*?*

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

Q: How to compute `$[(fact 'N)]`*?*

$$\texttt{\$[(fact 'N)]} \approx [\![\texttt{(fact 'N)}]\!]$$

```
⟦(fact 'N)⟧  = {1}∪{'N × $[(fact 'N)]}
```

Q: How to compute `$[(fact 'N)]`*?*

```
$[(fact 'N)] ≈ ⟦(fact 'N)⟧
```

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)
  (mrun ((fix-cache (Y (ev-cache ev))) e)))

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

```
(define (eval e)
  (mrun ((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* 0CFA

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 => Analyzer

Sound **Terminating** **Precise** **Extensible**