A Simple and Extensible Approach to Program Analysis

David Darais
University of Maryland
University of Vermont

Does my program cause a runtime error?

Does my program allocate too much?

Does my program sanitize all untrusted inputs?

Does my program have any data races?

🙁

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

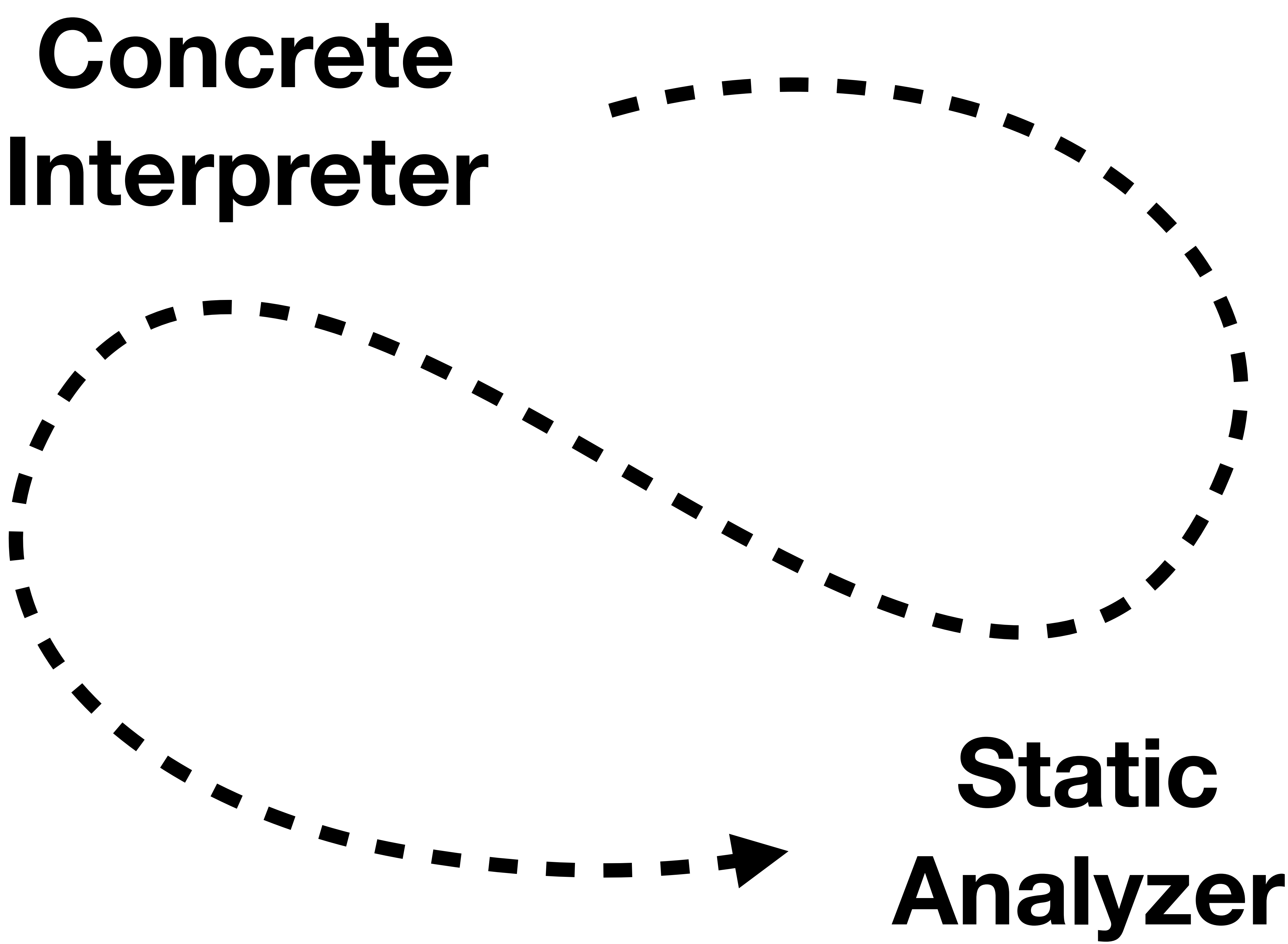
Sound **Terminating** **Precise** **Extensible**

Hypothesis:

It's easier to write a precise semantics than an abstract semantics.

Approach:

Write, maintain and debug one precise semantics.
Systematically derive multiple static analyzers.


Concrete
Interpreter
Static
Analyzer

Concrete Interpreter

```
if(N≠0){ x ≔ 100/N }
```

```
if(N≠0){ x ≔ 100/N }
```

N=1

```
if(N≠0){ x ≔ 100/N }
```

N=1

```
if(true){ x ≔ 100/N }
```

N=1

```
if(N≠0){ x ≔ 100/N }          N=1

if(true){ x ≔ 100/N }         N=1

         x ≔ 100/N            N=1
```

```
if(N≠0){ x ≔ 100/N }          N=1

if(true){ x ≔ 100/N }         N=1

      x ≔ 100/N               N=1

        100                   N=1  x=100
```

```
eval : exp × env ⇀ val × env
```

$$\mathtt{eval} : \mathtt{exp} \times \mathtt{env} \rightharpoonup \mathtt{val} \times \mathtt{env}$$

$$\mathtt{env} \coloneqq \mathtt{var} \rightharpoonup \mathtt{val}$$
$$\mathtt{val} \coloneqq \mathbb{B} \uplus \mathbb{Z}$$

$$\delta : \mathtt{op} \times \mathtt{val} \times \mathtt{val} \rightharpoonup \mathtt{val}$$

```
eval : exp × env ⇀ val × env
eval(Var(x),ρ) ≔ (ρ(x),ρ)
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

δ : op × val × val ⇀ val
```

```
eval : exp × env ⇀ val × env
eval(Var(x),ρ) ≔ (ρ(x),ρ)
eval(Assign(x,e),ρ) ≔
  (v,ρ′) ≔ eval(e,ρ)
  (v,ρ′[x↦v])
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

δ : op × val × val ⇀ val
```

```
eval : exp × env ⇀ val × env
eval(Var(x),ρ) ≔ (ρ(x),ρ)
eval(Assign(x,e),ρ) ≔
  (v,ρ′) ≔ eval(e,ρ)
  (v,ρ′[x↦v])
eval(Op(o,e₁,e₂),ρ) ≔
  (v₁,ρ′) ≔ eval(e₁,ρ)
  (v₂,ρ″) ≔ eval(e₂,ρ′)
  (δ(o,v₁,v₂),ρ″)
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

δ : op × val × val ⇀ val
```

```
eval : exp × env ⇀ val × env
eval(Var(x),ρ) ≔ (ρ(x),ρ)
eval(Assign(x,e),ρ) ≔
  (v,ρ′) ≔ eval(e,ρ)
  (v,ρ′[x↦v])
eval(Op(o,e₁,e₂),ρ) ≔
  (v₁,ρ′) ≔ eval(e₁,ρ)
  (v₂,ρ″) ≔ eval(e₂,ρ′)
  (δ(o,v₁,v₂),ρ″)
eval(If(e₁,e₂,e₃),ρ) ≔
  (v₁,ρ′) ≔ eval(e₁,ρ)
  cases
    v₁ = true  ⇒ eval(e₂,ρ′)
    v₁ = false ⇒ eval(e₃,ρ′)
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

δ : op × val × val ⇀ val
```

Concrete Interpreter

Monadic
Concrete
Interpreter

```
eval : exp × env ⇀ val × env
```

$$\mathtt{eval} : \mathtt{exp} \times \mathtt{env} \rightharpoonup \mathtt{val} \times \mathtt{env}$$

$$\approx$$

$$\mathtt{eval} : \mathtt{exp} \to M(\mathtt{val})$$

$$M(\mathtt{val}) \coloneqq \mathtt{env} \rightharpoonup \mathtt{val} \times \mathtt{env}$$

$$\mathsf{eval} : \mathsf{exp} \to M(\mathsf{val})$$

$$\mathsf{env} \coloneqq \mathsf{var} \rightharpoonup \mathsf{val}$$
$$\mathsf{val} \coloneqq \mathbb{B} \uplus \mathbb{Z}$$

$$\delta : \mathsf{op} \times \mathsf{val} \times \mathsf{val} \rightharpoonup \mathsf{val}$$

$$M(A) \coloneqq \mathsf{env} \rightharpoonup A \times \mathsf{env}$$

```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
```

```
 env ≔ var ⇀ val
 val ≔ 𝔹 ⊎ ℤ

   δ : op × val × val ⇀ val

M(A) ≔ env ⇀ A × env
```

```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

δ : op × val × val ⇀ val

M(A) ≔ env ⇀ A × env
```

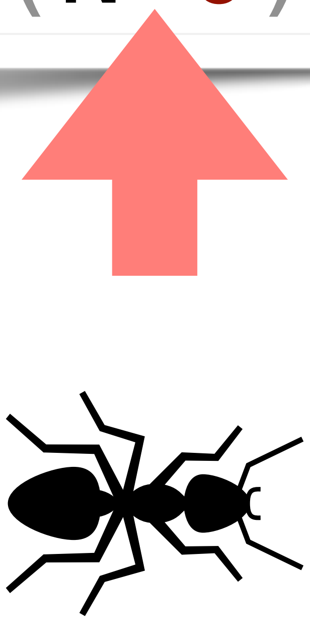
```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  return δ(o,v₁,v₂)
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval(e₁)
  cases
    v₁ = true  ⇒ eval(e₂)
    v₁ = false ⇒ eval(e₃)
```

```
env ≔ var ⇀ val
val ≔ 𝔹 ⊎ ℤ

  δ : op × val × val ⇀ val

M(A) ≔ env ⇀ A × env
```

```
if(N=0){ x := 100/N }
```



```
if(N=0){ x := 100/N }
```

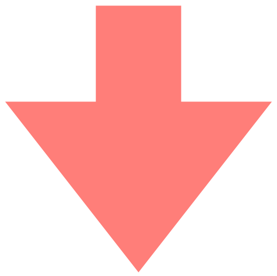
N=0

```
if(N=0){ x ≔ 100/N }
```

N=1

```
if(N=0){ x ≔ 100/N }
```

N=ANY



?

Monadic Concrete Interpreter

Monadic
Abstract
Interpreter

Abstract Values

$\mathbb{Z} \rhd \{-, 0, +\}$

$\mathbb{Z} \rhd \{-, 0, +\}$

$2 \ / \ (3 \ - \ 1)$

$$\mathbb{Z} \rhd \{-, 0, +\}$$

$$2 \;/\; (3 - 1)$$

$$\{+\} \;/\; (\{+\} - \{+\})$$

$$\mathbb{Z} \rhd \{-,0,+\}$$

$$\begin{array}{ccccccc} 2 & / & (& 3 & - & 1 &) \\ \{+\} & / & (& \{+\} & - & \{+\} &) \\ \{+\} & / & \{-,0,+\} & & & & \end{array}$$

$\mathbb{Z} \rhd \{-,0,+\}$

$$2 \;/\; (3 - 1)$$
$$\{+\} \;/\; (\{+\} - \{+\})$$
$$\{+\} \;/\; \{-,0,+\}$$

✓ $\{+,-\}$ *OR* ✗

```
eval : exp → M(val)
```

```
env ≔ var → val
val ≔ ℘(𝔹) ⊎ ℘({-,0,+})

  δ : op × val × val → val × 𝔹
```

Could the operation fail?

Abstract Values

Join Results

```
eval : exp → M(val)
```

```
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
```

```
env ≔ var → val
val ≔ ℘(𝔹) ⊎ ℘({-,0,+})

  δ : op × val × val → val × 𝔹
```

Abstract Values

Join Results

Variable Refinement

```
if(N≠0){ x ≔ 100/N }
```

N=ANY

```
if(N≠0){ x ≔ 100/N }
```

N=ANY

```
x ≔ 100/N
```

N∈{-,+}

```
eval : exp → M(val)

env ≔ var → val
val ≔ ℘(𝔹) ⊎ ℘({-,0,+})

  δ : op × val × val → val × 𝔹
⟦_⟧ : val → ℘(𝔹)
refine : exp × 𝔹 → M(void)

eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval(e₃)
```

```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval(e₃)
```

```
env ≔ var → val
val ≔ ℘(𝔹) ⊎ ℘({-,0,+})

     δ : op × val × val → val × 𝔹
   ⟦_⟧ : val → ℘(𝔹)
refine : exp × 𝔹 → M(void)
```

```
if(N≠0){ x ≔ 100/N }
```

N=ANY

```
while(true){}
```

<timeout>

```
fact(5)
```

<timeout>

Monadic Abstract Interpreter

Total
Monadic
Abstract
Interpreter

```
fact(ANY)
```

<timeout>

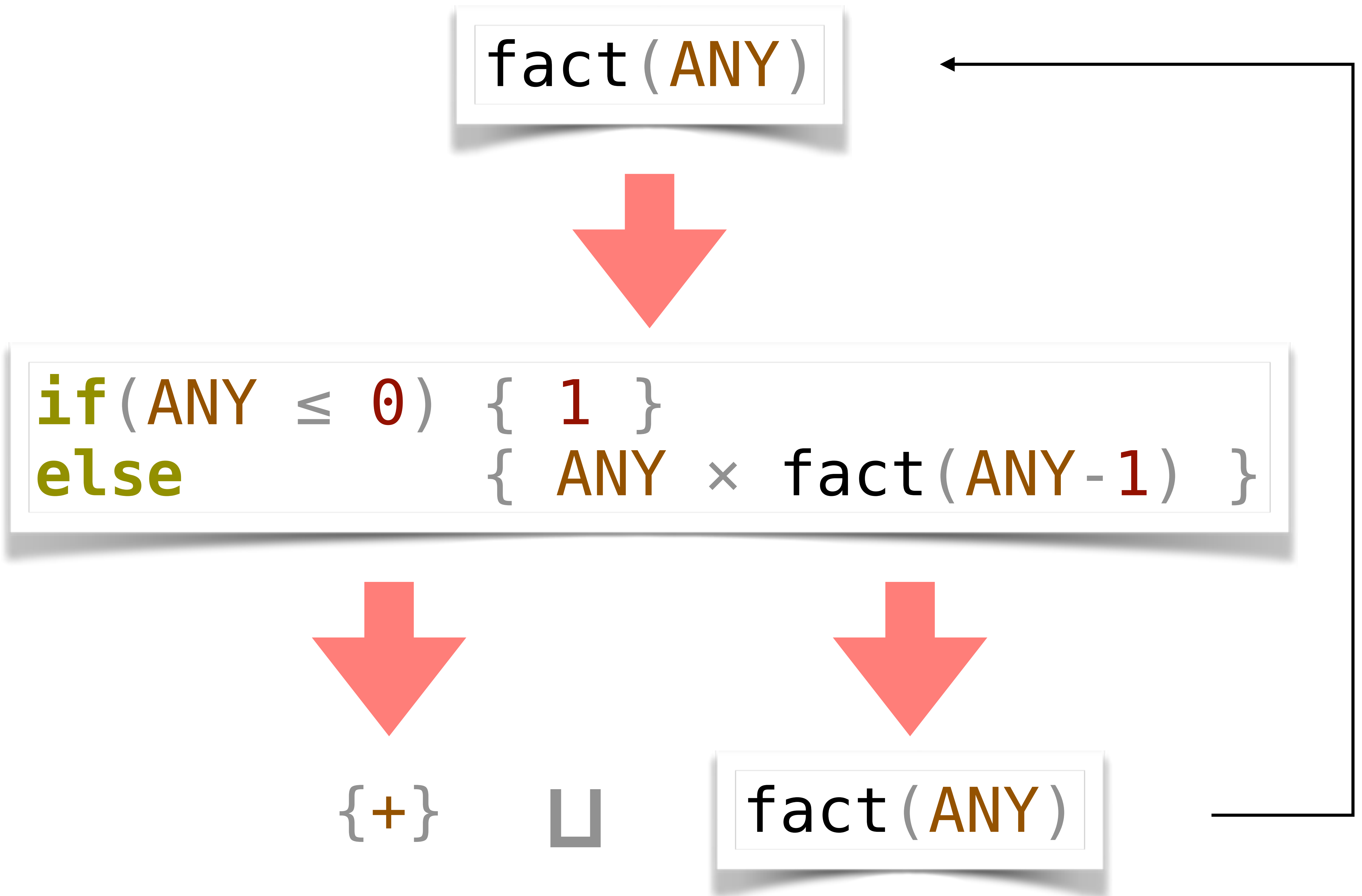
```
fact(ANY)
```

```
if(ANY ≤ 0) { 1 }
else        { ANY × fact(ANY-1) }
```

```
fact(ANY)
```

```
if(ANY ≤ 0) { 1 }
else        { ANY × fact(ANY-1) }
```

```
{+}
```


fact(ANY)
if(ANY ≤ 0) { 1 }
else { ANY × fact(ANY-1) }
{+}
⊔
fact(ANY)

$$⟦\mathtt{fact(ANY)}⟧ = \{+\} \sqcup ⟦\mathtt{fact(ANY)}⟧$$

$$[\![\mathtt{fact(ANY)}]\!] = \{+\} \sqcup [\![\mathtt{fact(ANY)}]\!]$$

$$[\![\mathtt{fact(ANY)}]\!] = \mathtt{lfp}(X).\ \{+\} \sqcup X$$

$$[\![\mathtt{fact(ANY)}]\!] = \{+\} \sqcup [\![\mathtt{fact(ANY)}]\!]$$

$$[\![\mathtt{fact(ANY)}]\!] = \mathtt{lfp(X).}\ \{+\} \sqcup \mathtt{X}$$

$$[\![\mathtt{fact(ANY)}]\!] = \{+\}$$

Q: How to teach interpreters to solve least-fixpoint equations between evaluation configurations and analysis results?

A: Caching

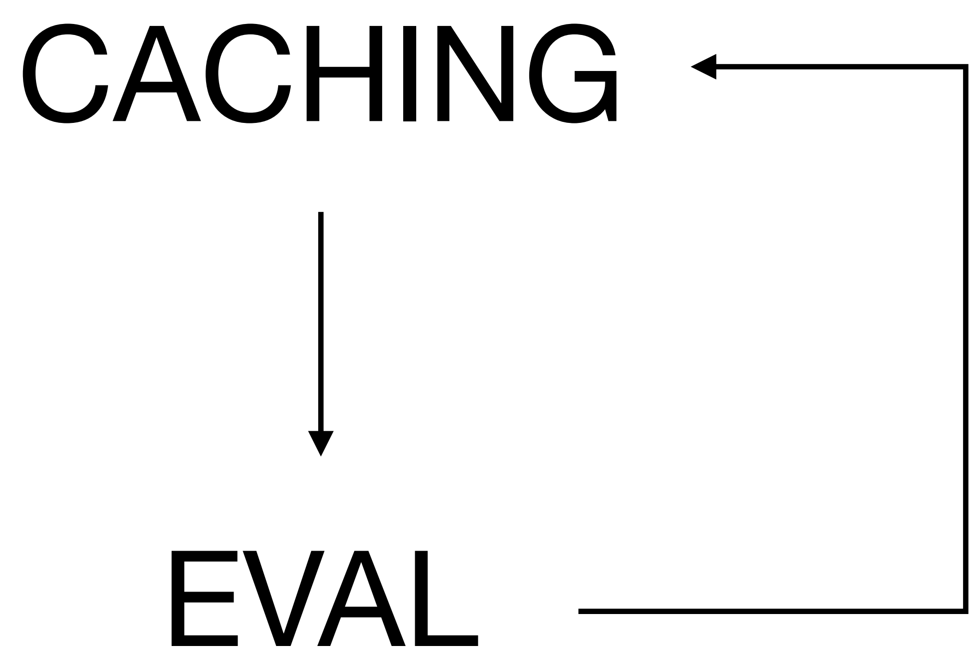
Darais, Labich, Nguyễn, Van Horn. Abstracting Definitional Interpreters. ICFP ’17.

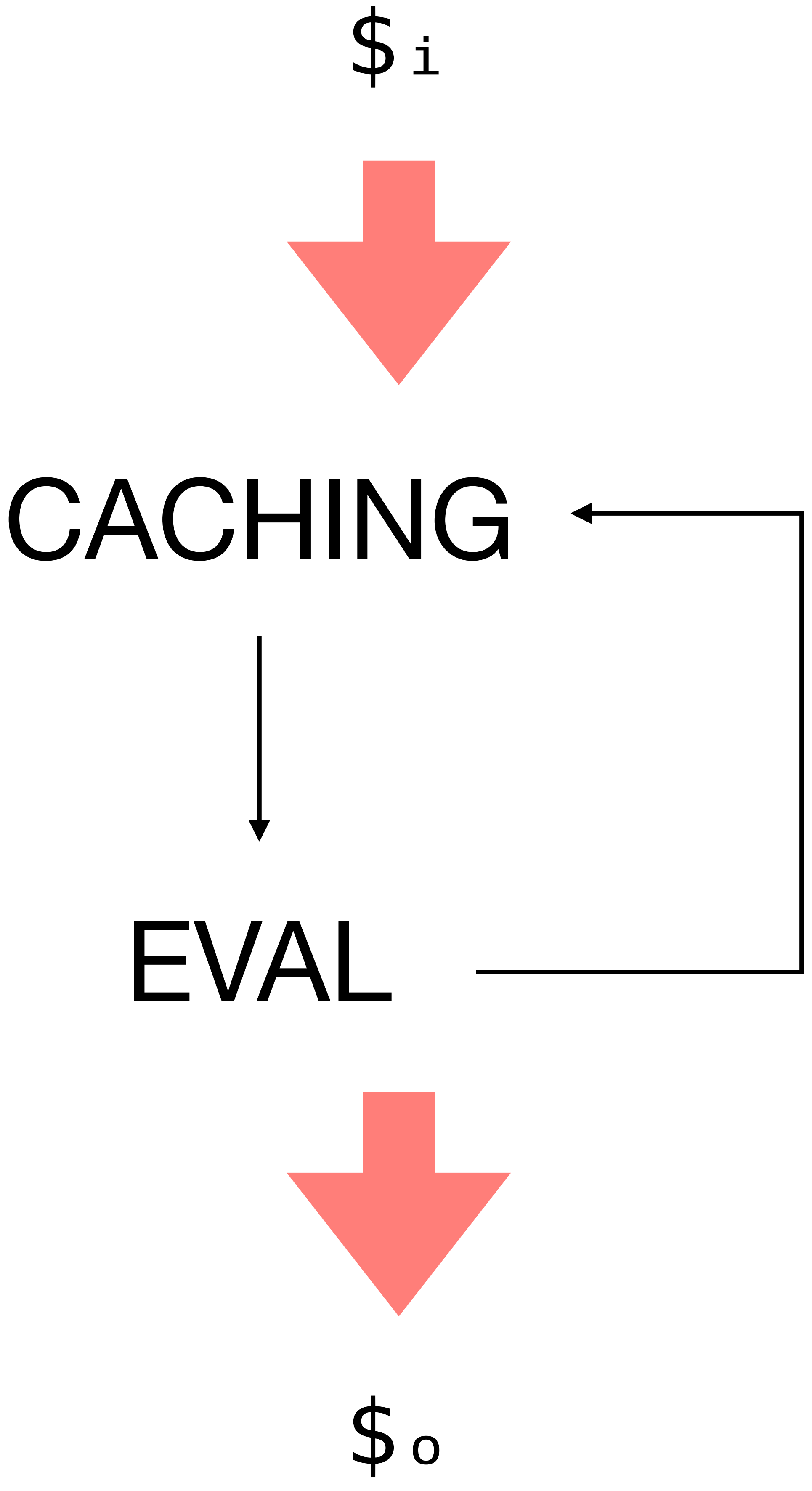
```
eval-cache : exp → M(val)
eval-cache(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
```

```
eval-cache : exp → M(val)
eval-cache(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

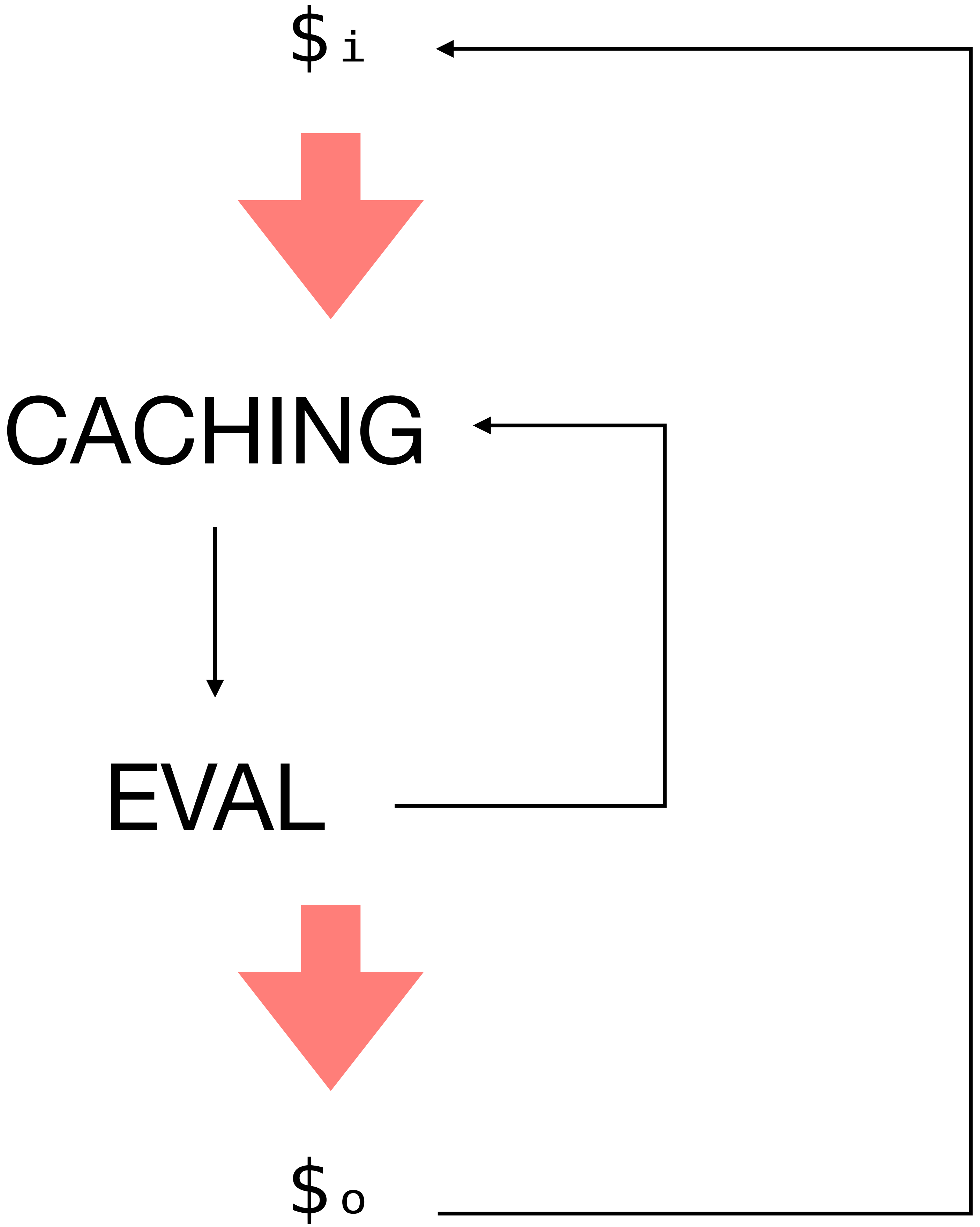
```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval-cache(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval-cache(e₁)
  v₂ ← eval-cache(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval-cache(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval-cache(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval-cache(e₃)

eval-cache : exp → M(val)
eval-cache(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```


CACHING
EVAL



$i
CACHING
EVAL
$o


$i
CACHING
EVAL
$o

$$\$_0(\mathtt{fact}(\mathtt{ANY})) \coloneqq \emptyset$$

$$\$_0(\mathtt{fact}(\mathtt{ANY})) \coloneqq \varnothing$$

$$\$_1(\mathtt{fact}(\mathtt{ANY})) \coloneqq \{+\}$$

$$\$_0(\mathrm{fact}(\mathrm{ANY})) \coloneqq \varnothing$$

$$\$_1(\mathrm{fact}(\mathrm{ANY})) \coloneqq \{+\}$$

$$\$_2(\mathrm{fact}(\mathrm{ANY})) \coloneqq \{+\}$$

$$\$_0(\mathrm{fact}(\mathrm{ANY})) \coloneqq \varnothing$$

$$\$_1(\mathrm{fact}(\mathrm{ANY})) \coloneqq \{+\}$$

$$\$_2(\mathrm{fact}(\mathrm{ANY})) \coloneqq \{+\}$$

✓

```
while(true){}
```

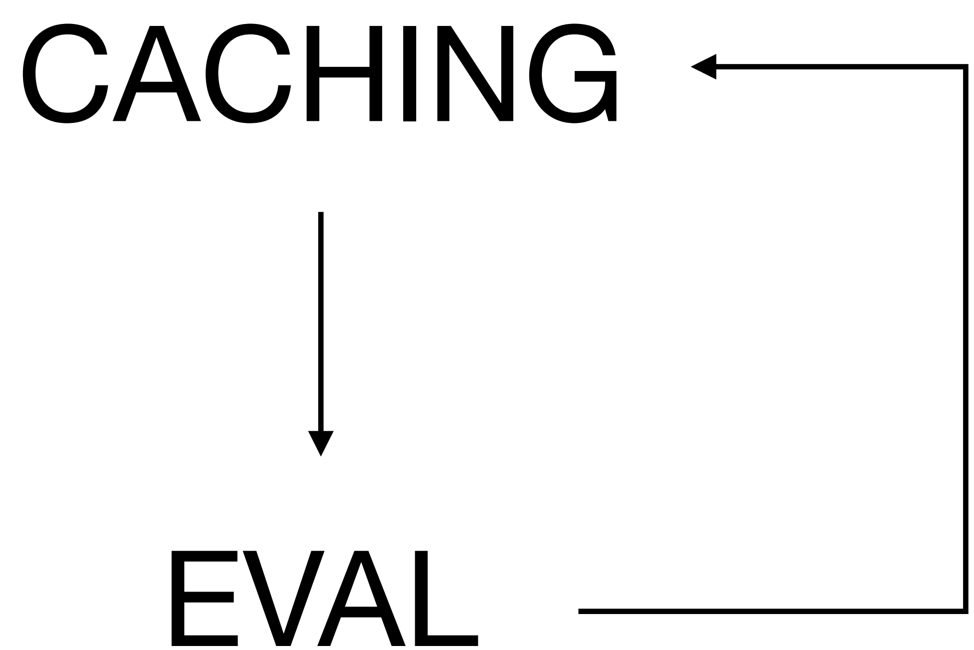
```
{}
```

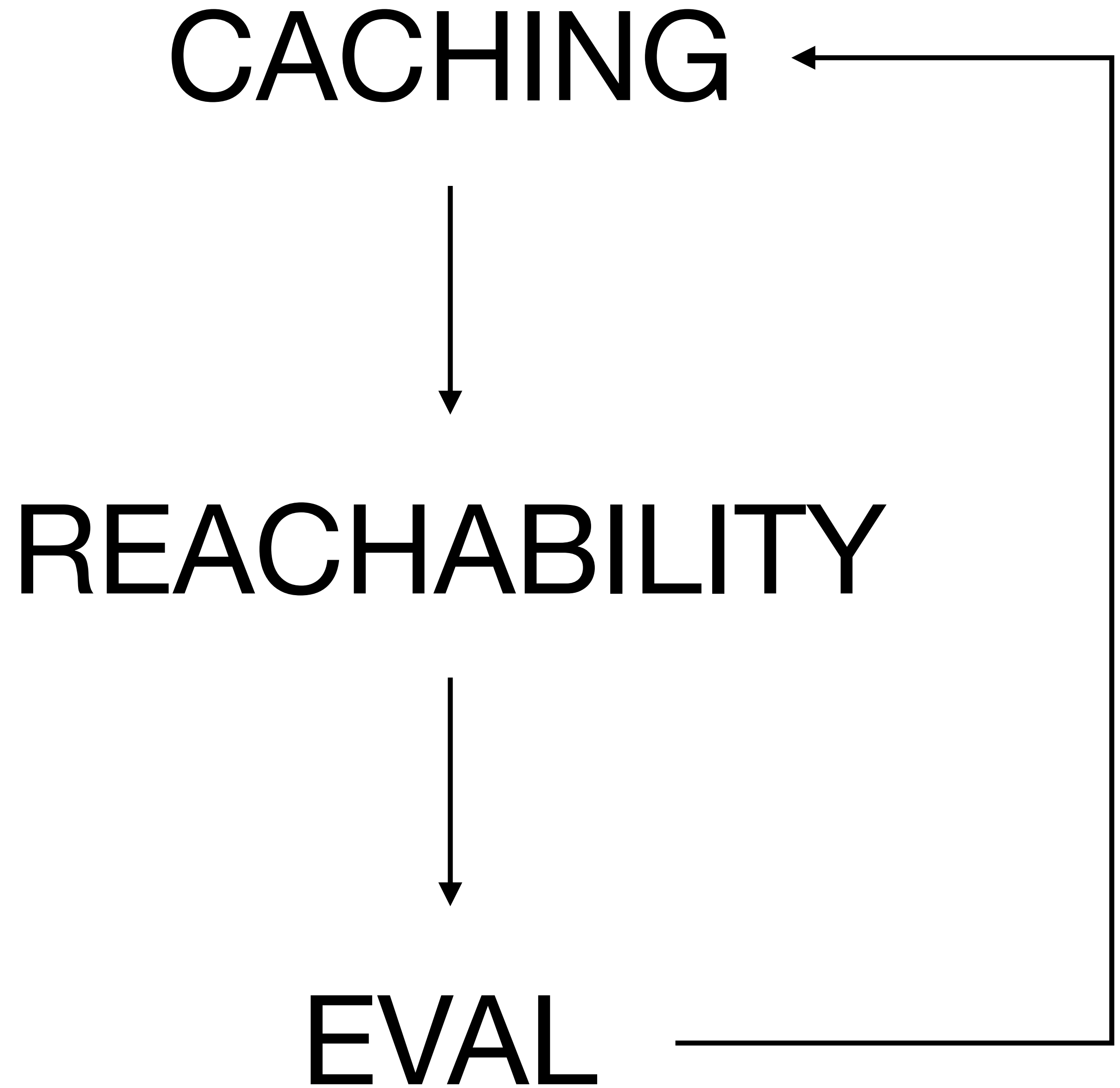
```
fact(ANY)
```

```
{+}
```

Total
Monadic
Abstract
Interpreter

Total
Monadic
Abstract
Extensible
Interpreter


CACHING
EVAL


CACHING
REACHABILITY
EVAL

“Unfixed” Interpreters

```
eval : exp → M(val)
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval-cache(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval-cache(e₁)
  v₂ ← eval-cache(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval-cache(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval-cache(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval-cache(e₃)
```

```
eval-cache : exp → M(val)
eval-cache(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

```
ev : (exp → M(val))
   → (exp → M(val))
eval(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
eval(Assign(x,e)) ≔ do
  v ← eval-cache(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
eval(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval-cache(e₁)
  v₂ ← eval-cache(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
eval(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval-cache(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval-cache(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval-cache(e₃)
```

```
ev-cache : (exp → M(val))
         → (exp → M(val))
eval-cache(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

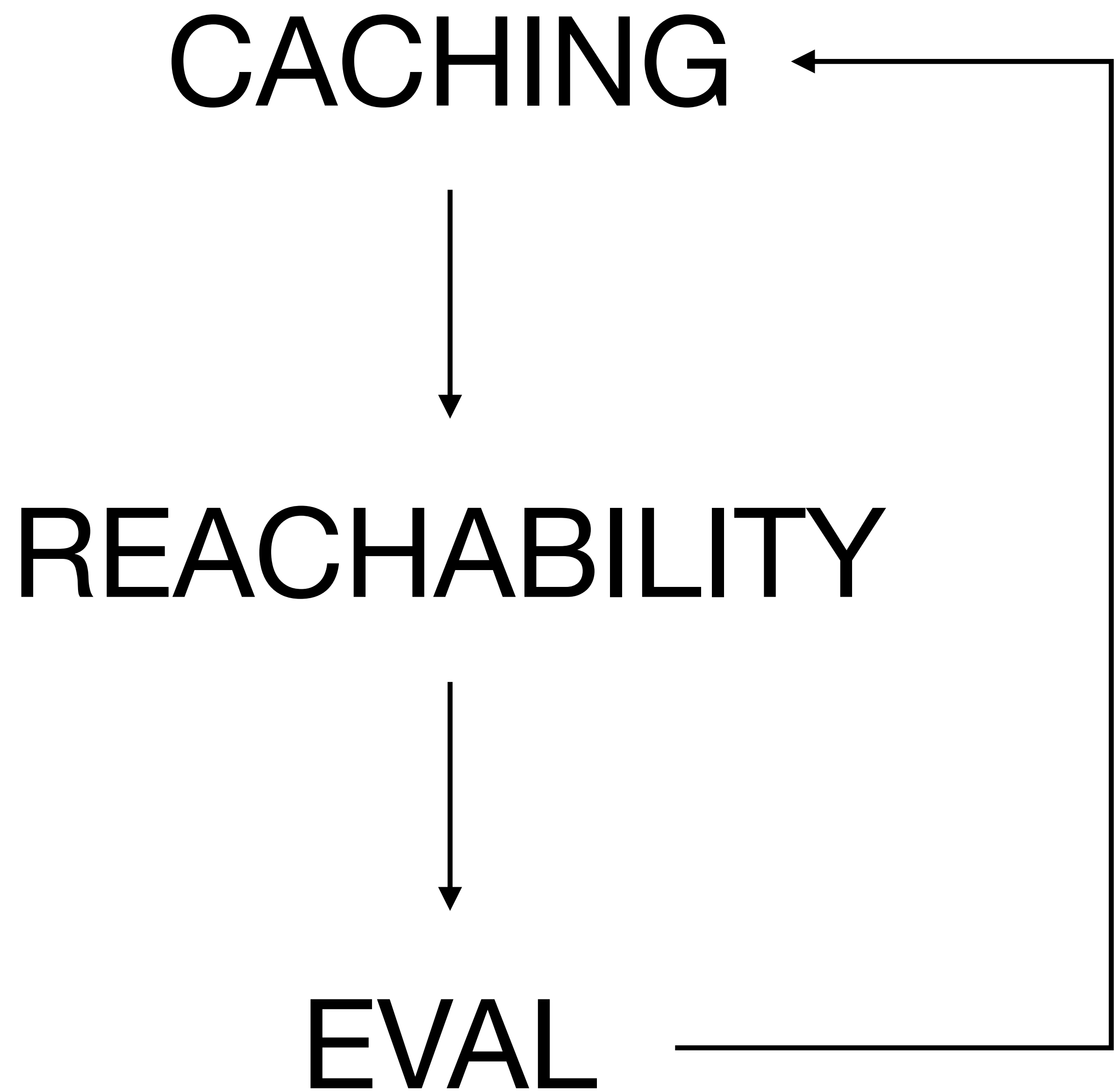
```
ev : (exp → M(val))
   → (exp → M(val))
ev(eval)(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
ev(eval)(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
ev(eval)(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v₃
ev(eval)(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval(e₁)
  join-cases
    ⟦v₁⟧ ∋ true ⇒ do
      refine(e₁,true)
      eval(e₂)
    ⟦v₁⟧ ∋ false ⇒ do
      refine(e₁,false)
      eval(e₃)
```

```
ev-cache : (exp → M(val))
         → (exp → M(val))
ev-cache(eval)(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

```
ev : (exp → M(val))
   → (exp → M(val))
ev(eval)(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
ev(eval)(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
ev(eval)(Op(o,e1,e2)) ≔ do
  v1 ← eval(e1)
  v2 ← eval(e2)
  (v3,err) ≔ δ(o,v1,v2)
  join-cases
    err = true ⇒ fail
    always     ⇒ return v3
ev(eval)(If(e1,e2,e3)) ≔ do
  v1 ← eval(e1)
  join-cases
    ⟦v1⟧ ∋ true ⇒ do
      refine(e1,true)
      eval(e2)
    ⟦v1⟧ ∋ false ⇒ do
      refine(e1,false)
      eval(e3)
```

```
ev-cache : (exp → M(val))
         → (exp → M(val))
ev-cache(eval)(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  else
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

```
ev-trace : (exp → M(val))
         → (exp → M(val))
ev-trace(eval)(e) ≔ do
  ρ ← get-env
  output-trace ⟨e,ρ⟩
  eval(e)
```




```
fix(ev-cache(ev-trace(ev)))
```

```
if(fact(N)≤0){expensive()}
```

N=ANY

dead = {expensive()}

```
ev : (exp → M(val))
   → (exp → M(val))
ev(eval)(Var(x)) ≔ do
  ρ ← get-env
  return ρ(x)
ev(eval)(Assign(x,e)) ≔ do
  v ← eval(e)
  ρ ← get-env
  put-env ρ[x↦v]
  return v
ev(eval)(Op(o,e₁,e₂)) ≔ do
  v₁ ← eval(e₁)
  v₂ ← eval(e₂)
  (v₃,err) ≔ δ(o,v₁,v₂)
  if(err) { fail }
  return v₃
ev(eval)(If(e₁,e₂,e₃)) ≔ do
  v₁ ← eval(e₁)
 join-cases
   ⟦v₁⟧ ∋ true ⇒ do
     refine(e₁,true)
     eval(e₂)
   ⟦v₁⟧ ∋ false ⇒ do
     refine(e₁,false)
     eval(e₃)
```

```
ev-cache : (exp → M(val))
         → (exp → M(val))
ev-cache(eval)(e) ≔ do
  ρ ← get-env
  if(seen(⟨e,ρ⟩))
  { return cached(⟨e,ρ⟩) }
  { mark-seen(⟨e,ρ⟩)
    v ← eval(e)
    update-cache(⟨e,ρ⟩ ↦ v) }
```

```
ev-trace : (exp → M(val))
         → (exp → M(val))
ev-trace(eval)(e) ≔ do
  ρ ← get-env
  output ⟨ρ,e⟩
  eval(e)
```

Sound

Terminating

Extensible

Path+Flow-Sensitive

Pushdown

Polarity-Numeric

Dead-code

Analysis

Sound

Terminating

Extensible

Path+Flow-Sensitive

Pushdown

Polarity-Numeric

Dead-code

Analysis

(context sensitivity)

(object sensitivity)

(path+flow sens)

(new numeric abs)

(objects+closures)

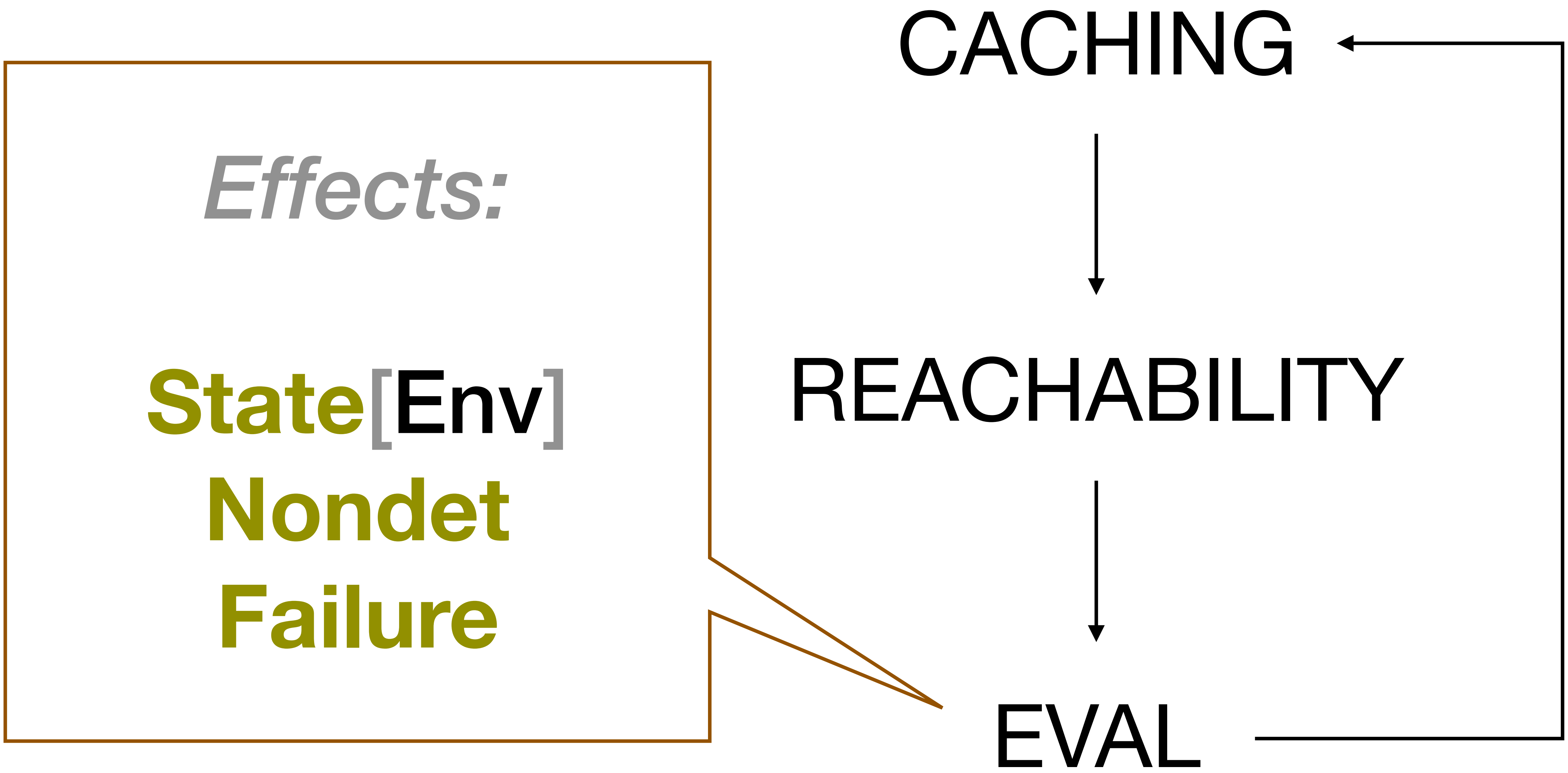
(symbolic execution)

...

Q: How to easily obtain variations in path and flow sensitivity for an analyzer.

A: Monads

Darais, Might, Van Horn.
Galois Transformers and Modular Abstract Interpreters.
OOPSLA '15.

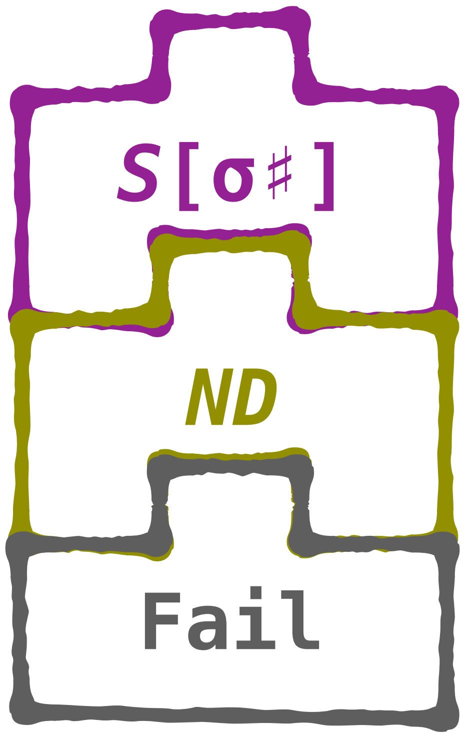



```
fix(ev-cache(ev-trace(ev)))
```

Effects:

State[Env]
Nondet
Failure

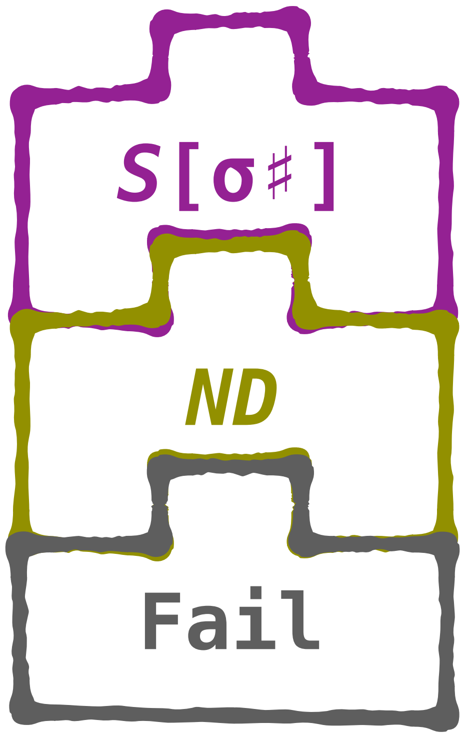
Monads:



Effects:

State[Env]
Nondet
Failure

Monads:

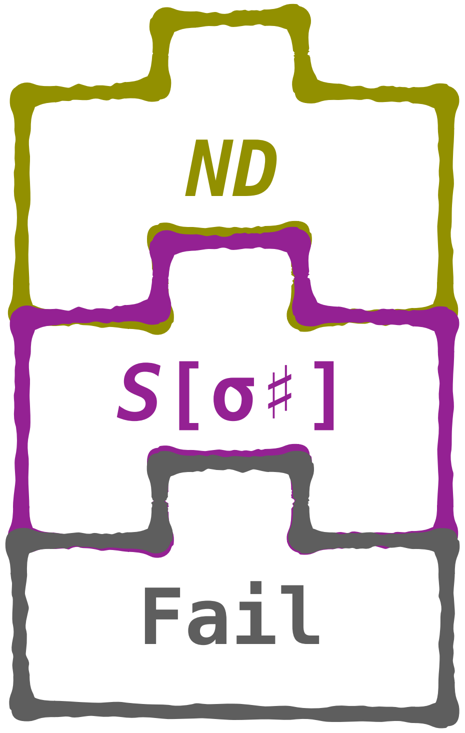


Path Sensitive

Effects:

State[Env]
Nondet
Failure

Monads:

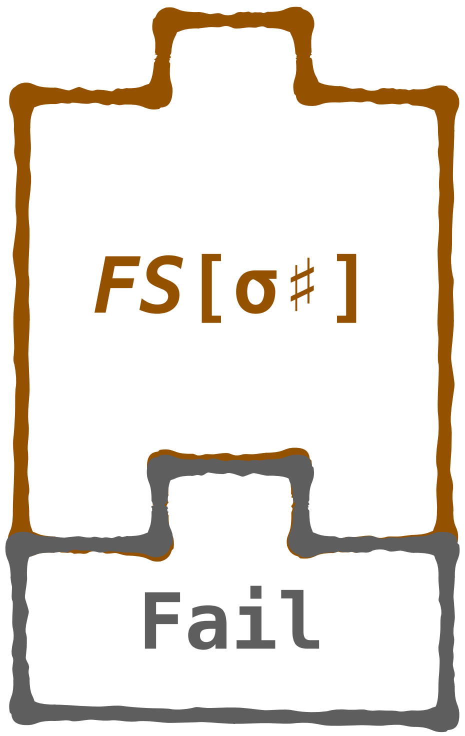


Flow Insensitive

Effects:

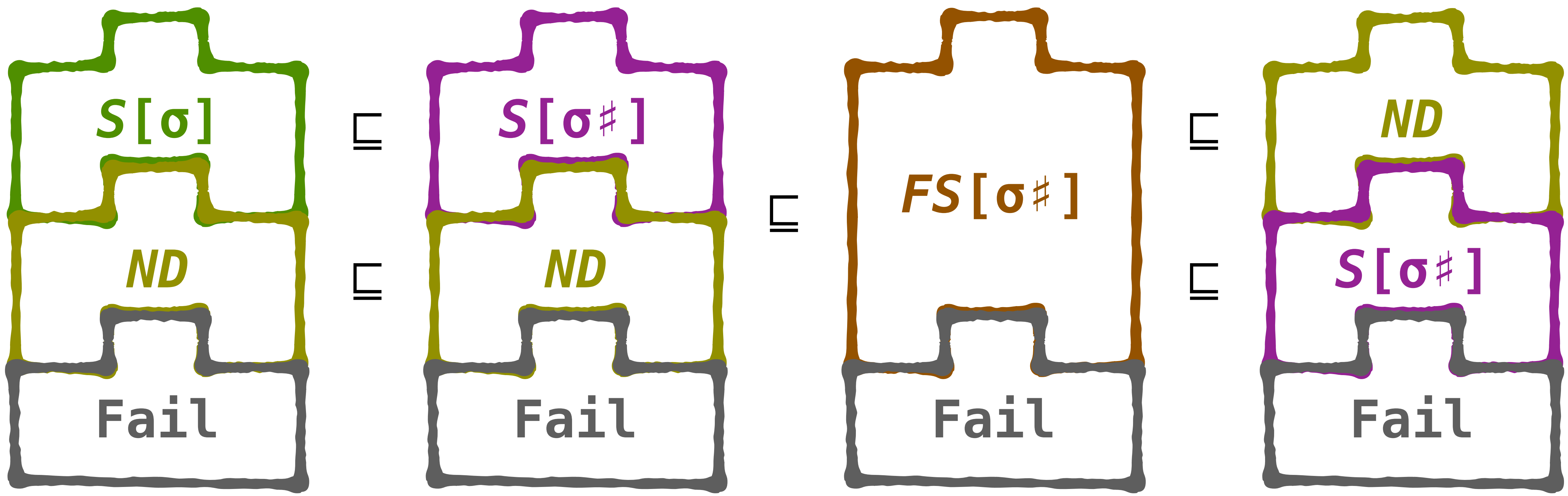
State[Env]
Nondet
Failure

Monads:



Flow Sensitive

Concrete Semantics	Path Sensitive	Flow Sensitive	Flow Insensitive



One Interpreter

More in the Papers

Soundness [OOPSLA ’15, ICFP ’17]

Pushdown Precision [ICFP ’17]

Sound Symbolic Execution [ICFP ’17]

Code Available in Haskell + Racket [OOPLA ’15,ICFP ’17]

😊

Go and Write Your Own Program Analyzer

It’s just a slightly fancy interpreter

Abstracting Definitional Interpreters

Interpreter => Analyzer

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