Environment Analysis of Higher-Order Languages

Matthew Might

Georgia Institute of Technology

7 June 2007

Thesis

Environment analysis is feasible and useful for higher-order languages.

Points

Environment analysis is **feasible**:

- Abstract counting. (ICFP 2006, VMCAI 2007, JFP 2007)
- Abstract frame strings. (POPL 2006, TCS 2007)
- Abstract garbage collection. (ICFP 2006, VMCAI 2007, JFP 2007)
- Configuration-widening, *etc.*

Environment analysis is **useful**:

- Super-β optimizations. (PLDI 2006)
- Logic-flow analysis. (POPL 2007)

These techniques are **novel**:

- Related work.

Why perform environment analysis?

- Globalization.
- Register-allocated environments.
- Lightweight closure conversion.
- Super-β inlining.
- Super-β copy propagation.
- Static closure allocation.
- Super-β rematerialization.
- Super-β teleportation.
- Escape analysis.
- Lightweight continuation conversion.
- Transducer fusion.
- Must-alias analysis.
- Logic-flow analysis & program verification.

Environments

An **environment** is a dictionary of names to values.

Environments

An **environment** is a dictionary of names to values.

Example

- $\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 4$
- $\mathtt{x} \mapsto$ `"foo"`

Environment facts

- May be created, extended, mutated, contracted and destroyed.
- Arbitrary number can arise during execution.

Environment problem (Take 1)

Given two environments, on which names do they agree in value?

Higher-order languages

A **higher-order language** allows computation/behavior as value.

Higher-order languages

A **higher-order language** allows computation/behavior as value.

Example

- Scheme/Lisp
- Standard ML

Higher-order languages

A **higher-order language** allows computation/behavior as value.

Example

- Scheme/Lisp
- Standard ML
- Java
- C++
- . . .

All face the same challenges in analysis.

The challenge: tri-facetted nature of λ

In one construct, λ is:

- control,
- environment,
- and data.

Outline

- Develop k-CFA.
- Formalize environment problem.
- Build abstract counting.
- Make it feasible: abstract garbage collection.
- Tour ΔCFA.
- Review applications.
- Look at related work.

What is k-CFA?

k-CFA

Where do λ terms flow?

Example

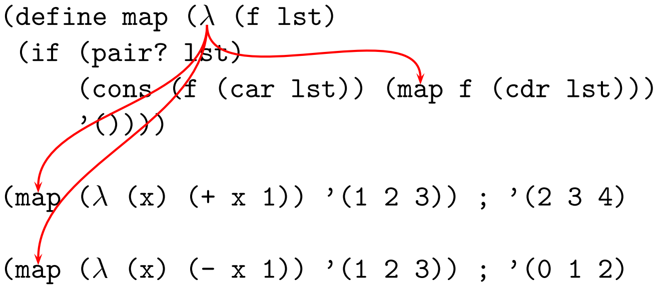
```
(define map (λ (f lst)
 (if (pair? lst)
     (cons (f (car lst)) (map f (cdr lst)))
     '())))

(map (λ (x) (+ x 1)) '(1 2 3)) ; '(2 3 4)

(map (λ (x) (- x 1)) '(1 2 3)) ; '(0 1 2)
```

What is k-CFA?

k-CFA

Where do λ terms flow?

Example



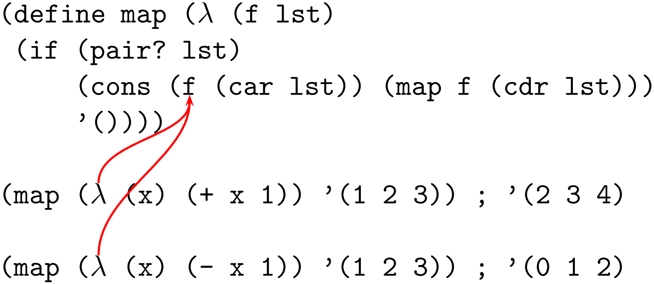
```
(define map (λ (f lst)
 (if (pair? lst)
     (cons (f (car lst)) (map f (cdr lst)))
     '())))


(map (λ (x) (+ x 1)) '(1 2 3)) ; '(2 3 4)

(map (λ (x) (- x 1)) '(1 2 3)) ; '(0 1 2)
```

What is k-CFA?

k-CFA

Where do λ terms flow?

Example



```
(define map (λ (f lst)
 (if (pair? lst)
     (cons (f (car lst)) (map f (cdr lst)))
     '())))


(map (λ (x) (+ x 1)) '(1 2 3)) ; '(2 3 4)


(map (λ (x) (- x 1)) '(1 2 3)) ; '(0 1 2)
```

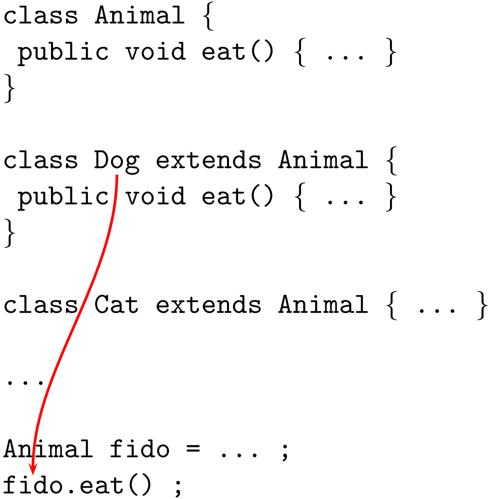
What is k-CFA?

Example

```
class Animal {
 public void eat() { ... }
}

class Dog extends Animal {
 public void eat() { ... }
}

class Cat extends Animal { ... }

...

Animal fido = ... ;
fido.eat() ;
```

What is k-CFA?

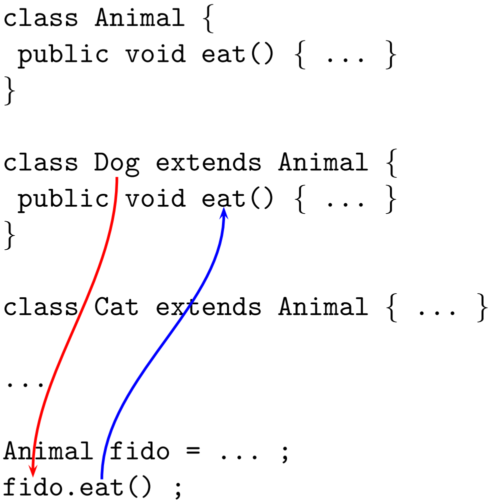
Example



```
class Animal {
 public void eat() { ... }
}

class Dog extends Animal {
 public void eat() { ... }
}

class Cat extends Animal { ... }

...

Animal fido = ... ;
fido.eat() ;
```

What is k-CFA?

Example



```
class Animal {
 public void eat() { ... }
}

class Dog extends Animal {
 public void eat() { ... }
}

class Cat extends Animal { ... }

...

Animal fido = ... ;
fido.eat() ;
```

What about environments?

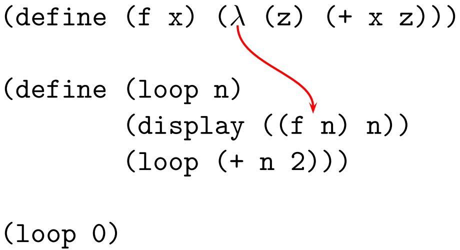
Closure = λ term + environment
Object = class + struct

```
(define (f x) (λ (z) (+ x z)))

(define (loop n)
        (display ((f n) n))
        (loop (+ n 2)))

(loop 0)
```

What about environments?

Closure = λ term + environment

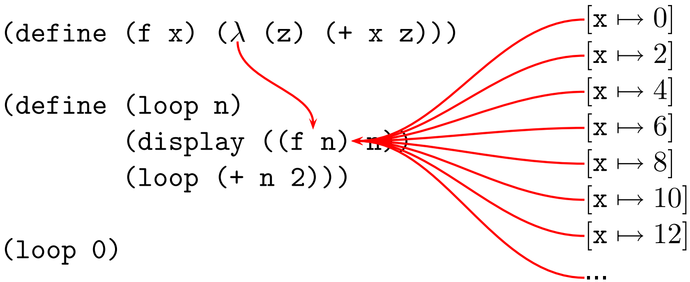
Object = class + struct



```
(define (f x) (λ (z) (+ x z)))

(define (loop n)
        (display ((f n) n))
        (loop (+ n 2)))

(loop 0)
```

What about environments?

Closure = λ term + environment
Object = class + struct

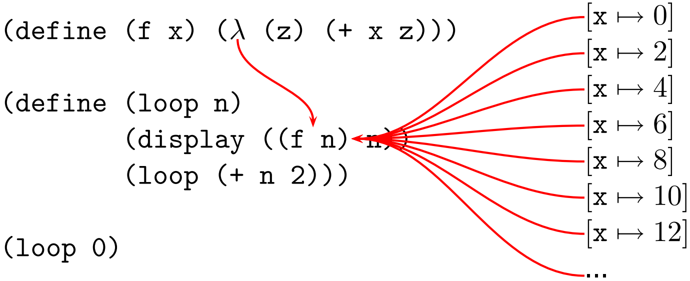


```
(define (f x) (λ (z) (+ x z)))

(define (loop n)
        (display ((f n) n))
        (loop (+ n 2)))

(loop 0)
```

$[\mathtt{x} \mapsto 0]$
$[\mathtt{x} \mapsto 2]$
$[\mathtt{x} \mapsto 4]$
$[\mathtt{x} \mapsto 6]$
$[\mathtt{x} \mapsto 8]$
$[\mathtt{x} \mapsto 10]$
$[\mathtt{x} \mapsto 12]$
...

What about environments?

Closure = λ term + environment
Object = class + struct



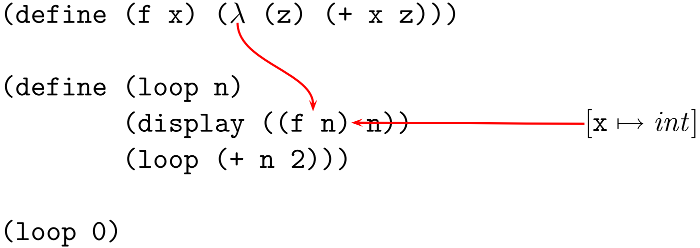
```
(define (f x) (λ (z) (+ x z)))

(define (loop n)
        (display ((f n) n))
        (loop (+ n 2)))

(loop 0)
```

$[x \mapsto 0]$
$[x \mapsto 2]$
$[x \mapsto 4]$
$[x \mapsto 6]$
$[x \mapsto 8]$
$[x \mapsto 10]$
$[x \mapsto 12]$
...

Environments must merge during finite analysis.

What about environments?

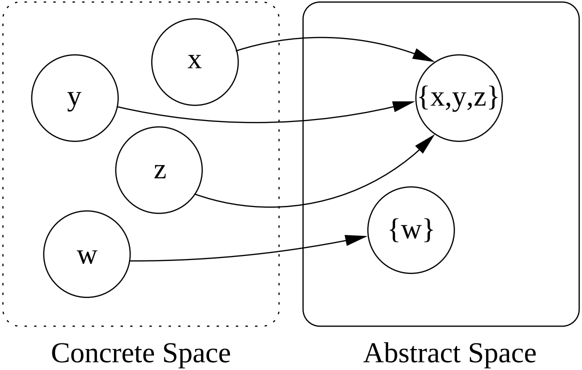
Closure = λ term + environment
Object = class + struct



```
(define (f x) (λ (z) (+ x z)))

(define (loop n)
        (display ((f n) n))
        (loop (+ n 2)))

(loop 0)
```

$[\mathtt{x} \mapsto int]$

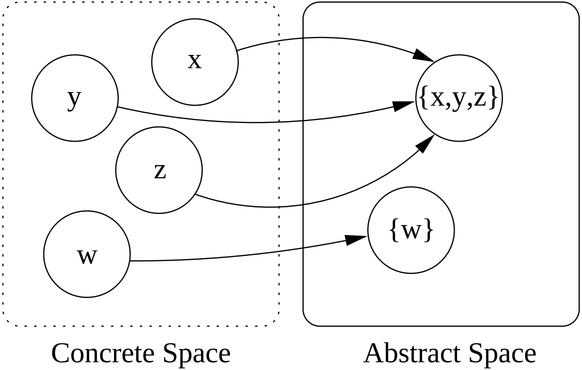
Problem: Merging blocks reasoning



Unsound reasoning

$|x| = |y|$, but $x \neq y$!

Problem: Merging blocks reasoning



Unsound reasoning

$|a| = |\mathrm{w}| = |b|$ *does* imply $a = b$.

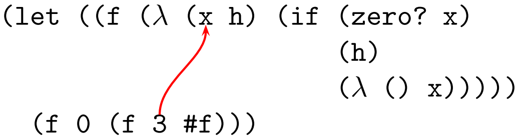
Problem: Inability to judge environments

```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments

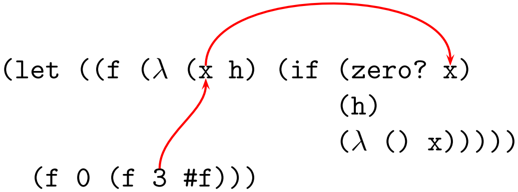


```
(let ((f (λ (x h) (if (zero? x)
                       (h)
                       (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments

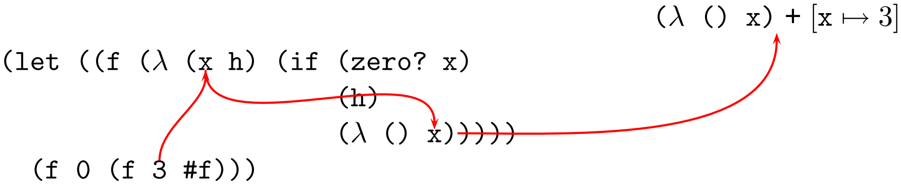


```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments



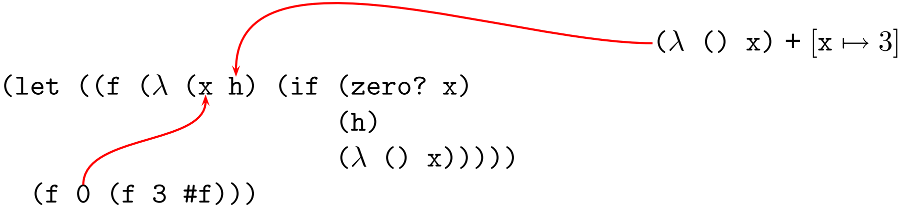
`(λ () x)` + $[\mathtt{x} \mapsto 3]$

```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments



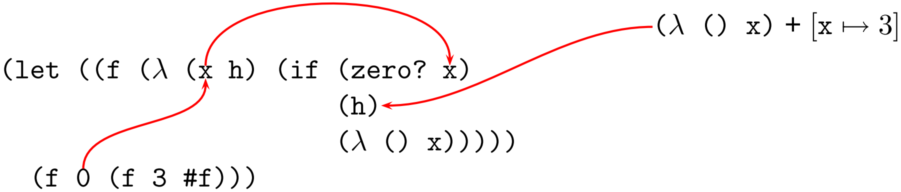
`(λ () x)` + $[\mathtt{x} \mapsto 3]$

```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments



$(\lambda\ ()\ \texttt{x}) + [\texttt{x} \mapsto 3]$

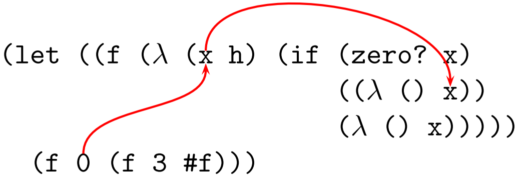
```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments

$(\lambda\ ()\ \texttt{x}) + [\texttt{x} \mapsto 3]$



```
(let ((f (λ (x h) (if (zero? x)
                      ((λ () x))
                      (λ () x)))))
  (f 0 (f 3 #f)))
```

Fact: `(λ () x)` flows to `(h)`.

Question: Safe to super-β inline?

Problem: Inability to judge environments

$(\lambda\ ()\ \mathtt{x}) + [\mathtt{x} \mapsto 3]$

```
(let ((f (λ (x h) (if (zero? x)
                       ((λ () 0))
                       (λ () 3)))))
  (f 0 (f 3 #f)))
```

Fact:	`(λ () x)` flows to `(h)`.
Question:	Safe to super-β inline?
Answer:	No.
Why:	Only *one* variable `x` in program; but *multiple dynamic bindings*.

Strategy

- Build k-CFA.
- Thread environment analysis through it.

Tool: Factored environments in k-CFA

$$\text{Env} = \text{Variable} \to \text{Value}$$

Tool: Factored environments in k-CFA

$$\text{Env} = \text{Variable} \rightarrow \lambda \times \text{Env}$$

Tool: Factored environments in k-CFA

$$\begin{tikzcd}[column sep=small] \text{Env} & = \text{Variable} \to \lambda \times & \text{Env} \arrow[ll, bend right] \end{tikzcd}$$

Must break recursion for analysis.

Tool: Factored environments in k-CFA

Env = Variable → Time

Tool: Factored environments in k-CFA

$$\text{Env} = \text{Variable} \rightarrow \text{Time}$$

$$\underbrace{\text{Variable} \times \text{Time}}_{\mathsf{Binding}} \rightarrow \text{Value}$$

Tool: Factored environments in k-CFA

$$\text{Env} = \text{Variable} \rightarrow \text{Time}$$

$$\underbrace{\text{Variable} \times \text{Time}}_{\text{Binding}} \rightarrow \text{Value}$$

Merge environments by partitioning Time into finite number of sets.

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: ς_1

Abstract: $\widehat{\varsigma}_1$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2$

Abstract: $\widehat{\varsigma}_1$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2 \longrightarrow \varsigma_3$

Abstract: $\widehat{\varsigma}_1$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete:

$$\begin{tikzcd}
\varsigma_1 \arrow[r] & \varsigma_2 \arrow[r] & \varsigma_3 \arrow[r] & \varsigma_4
\end{tikzcd}$$

Abstract:

$$\widehat{\varsigma}_1$$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2 \longrightarrow \varsigma_3 \longrightarrow \varsigma_4 \longrightarrow \varsigma_5 \longrightarrow \cdots$

Abstract: $\widehat{\varsigma}_1$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2 \longrightarrow \varsigma_3 \longrightarrow \varsigma_4 \longrightarrow \varsigma_5 \longrightarrow \cdots$

Abstract: $\widehat{\varsigma}_1 \longrightarrow \widehat{\varsigma}_2$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2 \longrightarrow \varsigma_3 \longrightarrow \varsigma_4 \longrightarrow \varsigma_5 \longrightarrow \cdots$

Abstract: $\widehat{\varsigma}_1 \longrightarrow \widehat{\varsigma}_2 \longrightarrow \widehat{\varsigma}_3$

Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $$\begin{tikzcd} \varsigma_1 \arrow[r] & \varsigma_2 \arrow[r] & \varsigma_3 \arrow[r] & \varsigma_4 \arrow[r] & \varsigma_5 \arrow[r] & \cdots \end{tikzcd}$$

Abstract: $$\begin{tikzcd} & & & \widehat{\varsigma}_{3.1,1} \\ \widehat{\varsigma}_1 \arrow[r] & \widehat{\varsigma}_2 \arrow[r] & \widehat{\varsigma}_3 \arrow[r] \arrow[ur] \arrow[dr] & \widehat{\varsigma}_4 \\ & & & \widehat{\varsigma}_{3.2,1} \end{tikzcd}$$

Tool: Abstract interpretation

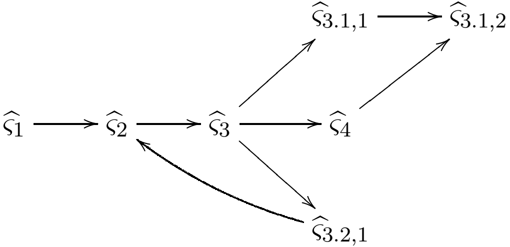
Definition

Abstract interpretation approximates set of reachable states.

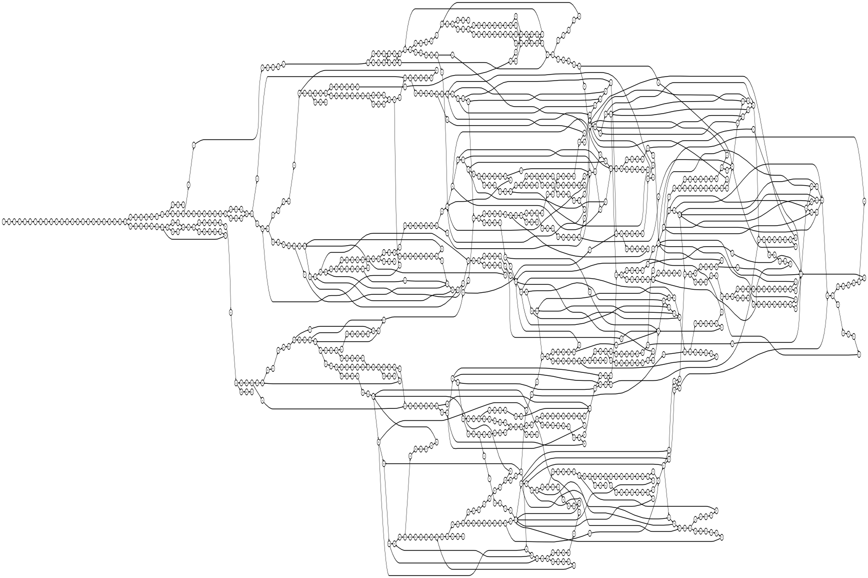
Interpretation

Concrete: $\varsigma_1 \longrightarrow \varsigma_2 \longrightarrow \varsigma_3 \longrightarrow \varsigma_4 \longrightarrow \varsigma_5 \longrightarrow \cdots$

Abstract:



Example



Tool: Continuation-passing style (CPS)

Contract

- Calls don’t return.
- Continuations (procedures) are passed—to receive return values.

Definition

A **continuation** encodes the future of computation.

Grammar

$$\begin{aligned} e, f \in \mathit{EXP} &::= v \\ &\quad\mid\ (\lambda\ (v_1 \cdots v_n)\ \mathit{call}) \\ \mathit{call} \in \mathit{CALL} &::= (f\ e_1 \cdots e_n) \end{aligned}$$

CPS narrows concern

λ is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ
conditional	call to λ
exception	call to λ
coroutine	call to λ
$\vdots$	$\vdots$

Advantage

Now λ is fine-grained construct.

Strategy

- Define state machine: $\varsigma \Rightarrow \varsigma'$.
- k-CFA = abstract interpretation of $\Rightarrow$.

Semantics: Eval states

$$\frac{}{(\qquad\qquad\qquad , \quad , \quad\; , \;)}$$

Semantics: Eval states

$$\frac{}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket,\quad ,\quad ,\ )}$$

Call site

Semantics: Eval states

$$\frac{}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, \quad , \quad)}$$

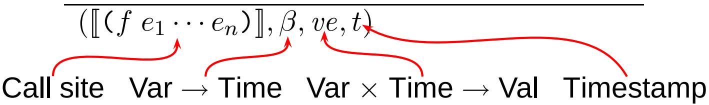
Call site   Var → Time

Semantics: Eval states

$$\overline{(\underbrace{[\![(f\ e_1 \cdots e_n)]\!]}_{\text{Call site}}, \underbrace{\beta}_{\text{Var} \to \text{Time}}, \underbrace{ve}_{\text{Var} \times \text{Time} \to \text{Val}}, \ \ )\qquad\qquad\qquad\qquad}$$

$$\mathcal{A}(v, \beta, ve) = \begin{cases} \textbf{let}\ t_{\text{bound}} & = \quad \beta(v) \\ \qquad\ \mathit{value} & = \quad ve(v, t_{\text{bound}}) \\ \ \ \textbf{in}\ \mathit{value} & \end{cases}$$

Semantics: Eval states



$$\frac{}{([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, t)}$$

Call site   Var → Time   Var × Time → Val   Timestamp

$$\mathcal{A}(v, \beta, ve) = \begin{cases} \textbf{let } t_{\text{bound}} & = & \beta(v) \\ \qquad \mathit{value} & = & ve(v, t_{\text{bound}}) \\ \textbf{ in } \mathit{value} & & \end{cases}$$

Semantics: Eval states

$$\frac{}{(\llbracket(f\ e_1 \cdots e_n)\rrbracket, \beta, ve, t) \Rightarrow (\qquad\ ,\ \ ,\ \ \ ,\qquad\ )}$$

$$\mathcal{A}(v, \beta, ve) = \left\{ \begin{array}{rcl} \textbf{let}\ t_{\text{bound}} & = & \beta(v) \\ \mathit{value} & = & ve(v, t_{\text{bound}}) \\ \textbf{in}\ \mathit{value} & & \end{array} \right.$$

Semantics: Eval states

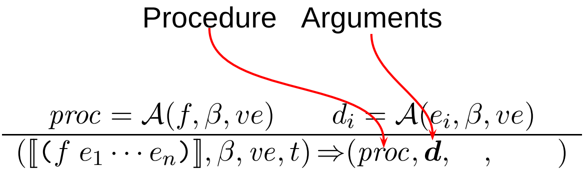
Procedure

$$\frac{proc = \mathcal{A}(f, \beta, ve)}{([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, t) \Rightarrow (proc, \quad , \quad , \qquad)}$$

$$\mathcal{A}(v, \beta, ve) = \left\{ \begin{array}{rcl} \textbf{let}\ t_{\text{bound}} & = & \beta(v) \\ value & = & ve(v, t_{\text{bound}}) \\ \textbf{in}\ value & & \end{array} \right.$$

$$\mathcal{A}(lam, \beta, ve) = (lam, \beta)$$

Semantics: Eval states


Procedure Arguments

$$\frac{proc = \mathcal{A}(f, \beta, ve) \qquad d_i = \mathcal{A}(e_i, \beta, ve)}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, \quad , \qquad)}$$

$$\mathcal{A}(v, \beta, ve) = \begin{cases} \textbf{let } t_{\text{bound}} & = \quad \beta(v) \\ \qquad value & = \quad ve(v, t_{\text{bound}}) \\ \ \ \textbf{in } value & \end{cases}$$

$$\mathcal{A}(lam, \beta, ve) = (lam, \beta)$$

Semantics: Eval states

Procedure Arguments

$$\frac{proc = \mathcal{A}(f, \beta, ve) \qquad d_i = \mathcal{A}(e_i, \beta, ve)}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, \qquad\quad)}$$

Var × Time → Val

$$\mathcal{A}(v, \beta, ve) = \begin{cases} \textbf{let } t_{\text{bound}} & = \quad \beta(v) \\ \qquad value & = \quad ve(v, t_{\text{bound}}) \\ \ \ \textbf{in } value & \end{cases}$$

$$\mathcal{A}(lam, \beta, ve) = (lam, \beta)$$

Semantics: Eval states

Procedure Arguments

$$\frac{proc = \mathcal{A}(f, \beta, ve) \qquad d_i = \mathcal{A}(e_i, \beta, ve)}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, t + 1)}$$

Var × Time → Val Timestamp

$$\mathcal{A}(v, \beta, ve) = \begin{cases} \textbf{let } t_{\text{bound}} & = & \beta(v) \\ \qquad value & = & ve(v, t_{\text{bound}}) \\ \ \textbf{in } value & & \end{cases}$$

$$\mathcal{A}(lam, \beta, ve) = (lam, \beta)$$

Semantics: Apply states

$$\frac{}{(\qquad\qquad\qquad\qquad , \quad , \quad\; , \;)}$$

Semantics: Apply states

$$\frac{}{((\llbracket\texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ \mathit{call}\texttt{)}\rrbracket, \beta'), \boldsymbol{d}, ve, t)}$$

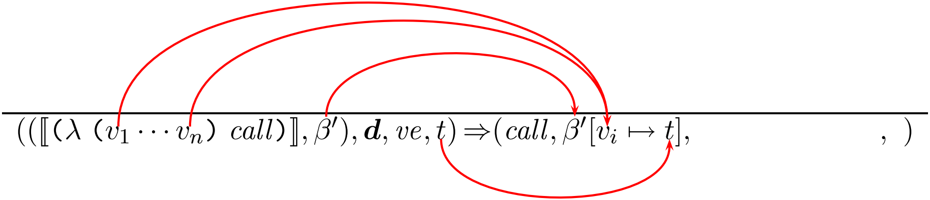
Semantics: Apply states

$$\frac{}{((\llbracket \texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)} \rrbracket, \beta'), \boldsymbol{d}, ve, t) \Rightarrow (\qquad , \qquad\qquad\qquad , \qquad\qquad\qquad\qquad , \ )}$$

Semantics: Apply states

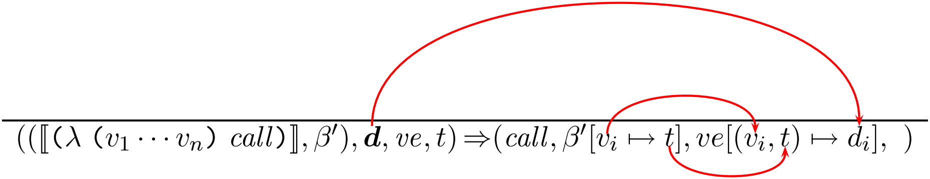
$$\frac{}{((\llbracket\texttt{(}\lambda\ \texttt{(}v_1\cdots v_n\texttt{)}\ call\texttt{)}\rrbracket, \beta'), \boldsymbol{d}, ve, t) \Rightarrow (call, \qquad\qquad , \qquad\qquad\qquad\qquad , \ )}$$

Semantics: Apply states



$$\frac{}{((\llbracket \texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)} \rrbracket, \beta'), \boldsymbol{d}, ve, t) \Rightarrow (call, \beta'[v_i \mapsto t], \qquad\qquad\qquad , \ )}$$

Semantics: Apply states



$$\frac{}{((\llbracket\texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)}\rrbracket, \beta'), \boldsymbol{d}, ve, t) \Rightarrow (call, \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d_i],\ \ )}$$

Semantics: Apply states

$$\frac{}{((\llbracket \texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ \mathit{call}\texttt{)} \rrbracket, \beta'), \boldsymbol{d}, ve, t) \Rightarrow (\mathit{call}, \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d_i], t)}$$

Eval-state transition

$$\frac{proc = \mathcal{A}(f, \beta, ve) \qquad d_i = \mathcal{A}(e_i, \beta, ve)}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, t+1)}$$

Apply-state transition

$$\frac{proc = (\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \beta')}{(proc, \boldsymbol{d}, ve, t) \Rightarrow (call, \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d_i], t)}$$

Domains

$$\begin{array}{rclcl}
\varsigma & \in & Eval & = & CALL \times BEnv \times VEnv \times Time \\
 & + & Apply & = & Proc \times D^* \times VEnv \times Time \\
\beta & \in & BEnv & = & VAR \rightarrow Time \\
ve & \in & VEnv & = & VAR \times Time \rightarrow D \\
proc & \in & Proc & = & Clo + \{halt\} \\
clo & \in & Clo & = & LAM \times BEnv \\
d & \in & D & = & Proc \\
t & \in & Time & = & \text{infinite set of times (contours)}
\end{array}$$

Lookup function

$$\mathcal{A}(lam, \beta, ve)$$
$$= (lam, \beta)$$
$$\mathcal{A}(v, \beta, ve)$$
$$= ve(v, \beta(v))$$

Eval-state transition

$$\frac{\widehat{proc} \in \widehat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve}) \qquad \hat{d}_i = \widehat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve})}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{t}) \gtrapprox (\widehat{proc}, \hat{\mathbf{d}}, \widehat{ve}, \widehat{succ}(\hat{t}))}$$

Apply-state transition

$$\frac{\widehat{proc} = (\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \hat{\beta}')}{(\widehat{proc}, \hat{\mathbf{d}}, \widehat{ve}, \hat{t}) \gtrapprox (call, \hat{\beta}'[v_i \mapsto \hat{t}], \widehat{ve} \sqcup [(v_i, \hat{t}) \mapsto \hat{d}_i], \hat{t})}$$

Domains

$$\begin{array}{rcrcl}
\hat{\varsigma} & \in & \widehat{Eval} & = & CALL \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\
 & + & \widehat{Apply} & = & \widehat{Proc} \times \hat{D}^* \times \widehat{VEnv} \times \widehat{Time} \\
\hat{\beta} & \in & \widehat{BEnv} & = & VAR \rightarrow \widehat{Time} \\
\widehat{ve} & \in & \widehat{VEnv} & = & VAR \times \widehat{Time} \rightarrow \hat{D} \\
\widehat{proc} & \in & \widehat{Proc} & = & \widehat{Clo} + \{halt\} \\
\widehat{clo} & \in & \widehat{Clo} & = & LAM \times \widehat{BEnv} \\
\hat{d} & \in & \hat{D} & = & \mathcal{P}(\widehat{Proc}) \\
\hat{t} & \in & \widehat{Time} & = & \text{finite set of times (contours)}
\end{array}$$

Lookup function

$$\widehat{\mathcal{A}}(lam, \hat{\beta}, \widehat{ve})$$

$$= \{(lam, \hat{\beta})\}$$

$$\widehat{\mathcal{A}}(v, \hat{\beta}, \widehat{ve})$$

$$= \widehat{ve}(v, \hat{\beta}(v))$$

Eval-state transition

$$\frac{\widehat{proc} \in \widehat{\mathcal{A}}(f, \widehat{\beta}, \widehat{ve}) \qquad \widehat{d}_i = \widehat{\mathcal{A}}(e_i, \widehat{\beta}, \widehat{ve})}{(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{t}) \mathrel{\approx\!\!>} (\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{ve}, \widehat{succ}(\widehat{t}))}$$

Apply-state transition

$$\frac{\widehat{proc} = (\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \widehat{\beta}')}{(\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{ve}, \widehat{t}) \mathrel{\approx\!\!>} (call, \widehat{\beta}'[v_i \mapsto \widehat{t}], \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i], \widehat{t})}$$

Domains

$$\begin{array}{rcccl}
\widehat{\varsigma} & \in & \widehat{Eval} & = & CALL \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \\
 & + & \widehat{Apply} & = & \widehat{Proc} \times \widehat{D}^* \times \widehat{VEnv} \times \widehat{Time} \\
\widehat{\beta} & \in & \widehat{BEnv} & = & VAR \rightarrow \widehat{Time} \\
\widehat{ve} & \in & \widehat{VEnv} & = & VAR \times \widehat{Time} \rightarrow \widehat{D} \\
\widehat{proc} & \in & \widehat{Proc} & = & \widehat{Clo} + \{halt\} \\
\widehat{clo} & \in & \widehat{Clo} & = & LAM \times \widehat{BEnv} \\
\widehat{d} & \in & \widehat{D} & = & \mathcal{P}(\widehat{Proc}) \\
\widehat{t} & \in & \widehat{Time} & = & \text{finite set of times (contours)}
\end{array}$$

Lookup function

$$\widehat{\mathcal{A}}(lam, \widehat{\beta}, \widehat{ve})$$

$$= \{(lam, \widehat{\beta})\}$$

$$\widehat{\mathcal{A}}(v, \widehat{\beta}, \widehat{ve})$$

$$= \widehat{ve}(v, \widehat{\beta}(v))$$

Environment analysis, Take 1: μCFA

Environment problem refined

Input

Two abstract environments, $\widehat{\beta}_1$ and $\widehat{\beta}_2$.

Environment problem refined

Input

Two abstract environments, $\widehat{\beta}_1$ and $\widehat{\beta}_2$.

Output

The set of variables on which their concrete counterparts agree.

Strategy

- Count concrete counterparts to abstract bindings.

Strategy

- Count concrete counterparts to abstract bindings.
- Apply principle: $\{x\} = \{y\} \implies x = y$.

Tool: Abstract counting

Abstract binding counter, $\widehat{\mu}$: “Bindings” $\rightarrow \{0, 1, \infty\}$.

Eval

$$(\llbracket \texttt{(}f\ e_1 \cdots e_n\texttt{)} \rrbracket, \widehat{\beta}, \widehat{ve}, \quad \widehat{t}) \approx\!\!> (\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{ve}, \quad \widehat{succ}(\widehat{t}))$$

$$\text{where} \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}(f, \widehat{\beta}, \widehat{ve}) \\ \widehat{d}_i = \widehat{\mathcal{A}}(e_i, \widehat{\beta}, \widehat{ve}) \end{cases}$$

Apply

$$((\llbracket \texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)} \rrbracket, \widehat{\beta}_b), \widehat{\boldsymbol{d}}, \widehat{ve}, \quad \widehat{t}) \approx\!\!> (call, \widehat{\beta}', \widehat{ve}', \quad \widehat{t})$$

$$\text{where} \begin{cases} \widehat{\beta}' = \widehat{\beta}_b[v_i \mapsto \widehat{t}] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i] \\ \end{cases}$$

Tool: Abstract counting

Abstract binding counter, $\widehat{\mu}$: “Bindings” $\to \{0, 1, \infty\}$.

Eval

$$([\![\texttt{(}f\ e_1 \cdots e_n\texttt{)}]\!], \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{ve}, \widehat{\mu}, \widehat{succ}(\widehat{t}))$$

$$\text{where} \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}(f, \widehat{\beta}, \widehat{ve}) \\ \widehat{d}_i = \widehat{\mathcal{A}}(e_i, \widehat{\beta}, \widehat{ve}) \end{cases}$$

Apply

$$(([\![\texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)}]\!], \widehat{\beta}_b), \widehat{\boldsymbol{d}}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (call, \widehat{\beta}', \widehat{ve}', \widehat{\mu}', \widehat{t})$$

$$\text{where} \begin{cases} \widehat{\beta}' = \widehat{\beta}_b[v_i \mapsto \widehat{t}] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i] \\ \widehat{\mu}' = \widehat{\mu} \oplus [(v_i, \widehat{t}) \mapsto 1] \end{cases}$$

μCFA environment condition

Basic Principle

If $\{x\} = \{y\}$, then $x = y$.

Theorem (Environment condition)

If $\widehat{\beta}_1(v) = \widehat{\beta}_2(v)$,

and $\widehat{\mu}(v, \widehat{\beta}_1(v)) = \widehat{\mu}(v, \widehat{\beta}_2(v)) = 1$,

then $\beta_1(v) = \beta_2(v)$.

μCFA environment condition

Basic Principle

If $\{x\} = \{y\}$, then $x = y$.

Theorem (Environment condition)

If $\widehat{\beta}_1(v) = \widehat{\beta}_2(v)$,

and $\widehat{\mu}(v, \widehat{\beta}_1(v)) = \widehat{\mu}(v, \widehat{\beta}_2(v)) = 1$,

then $\beta_1(v) = \beta_2(v)$,

where: $(v, \beta_1(v)) \in dom(ve)$,

and $|\beta_i| \sqsubseteq \widehat{\beta}_i$,

and $|ve|^{\mu} \sqsubseteq \widehat{\mu}$.

μCFA environment condition

Basic Principle

If $\{x\} = \{y\}$, then $x = y$.

Theorem (Environment condition)

If $\widehat{\beta}_1(v) = \widehat{\beta}_2(v)$,

and $\widehat{\mu}(v, \widehat{\beta}_1(v)) = \widehat{\mu}(v, \widehat{\beta}_2(v)) = 1$,

then $\beta_1(v) = \beta_2(v)$,

where: $(v, \beta_1(v)) \in dom(ve)$,

and $|\beta_i| \sqsubseteq \widehat{\beta}_i$,

and $|ve|^{\mu} \sqsubseteq \widehat{\mu}$.

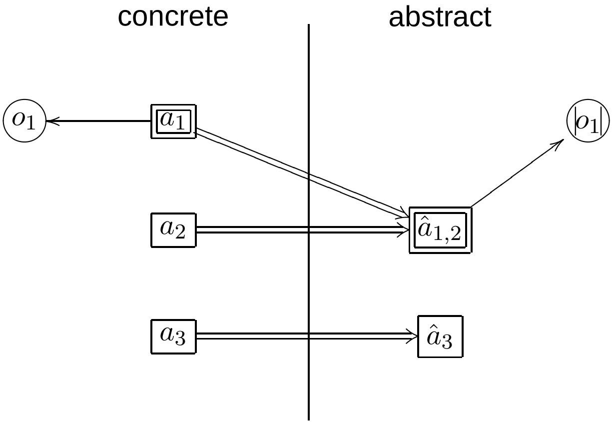
Problem

Most counts hit ∞: almost every variable bound more than once!

Making it feasible: ΓCFA

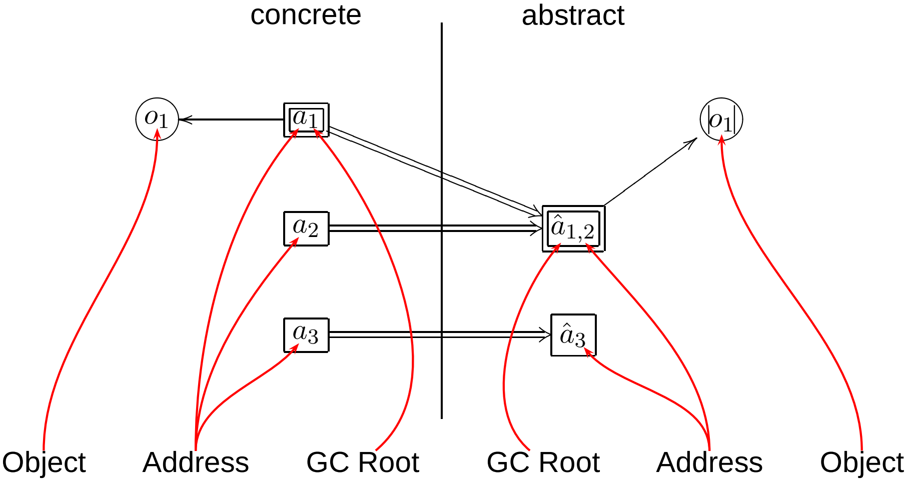
Example: Abstract garbage collection

3-address concrete heap. 2-address abstract counterpart.



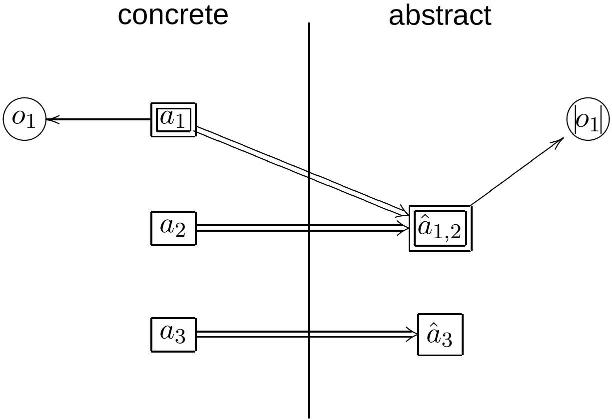
Example: Abstract garbage collection

3-address concrete heap. 2-address abstract counterpart.



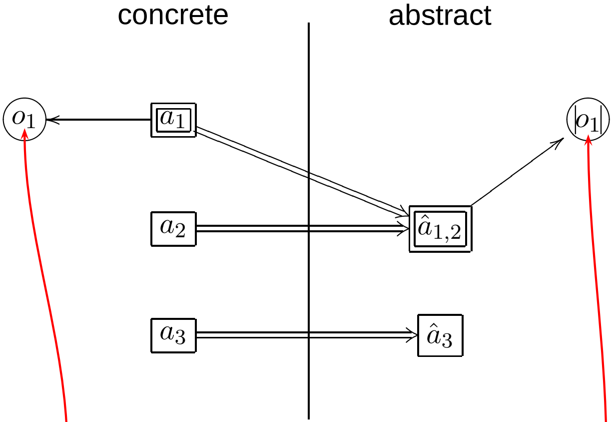
Example: Abstract garbage collection

3-address concrete heap. 2-address abstract counterpart.



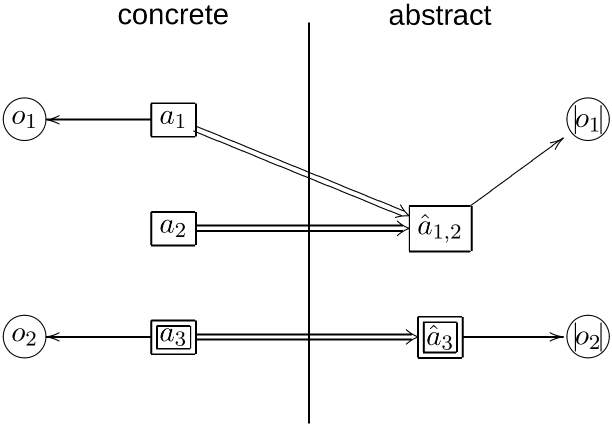
Example: Abstract garbage collection

Next: Allocate object o_2 to address a_3. Shift root to a_3.



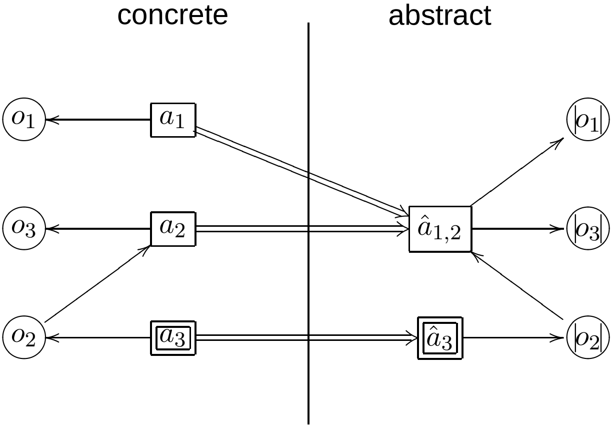
Example: Abstract garbage collection

Next: Allocate object o_3 to address a_2. Point o_2 to a_2.



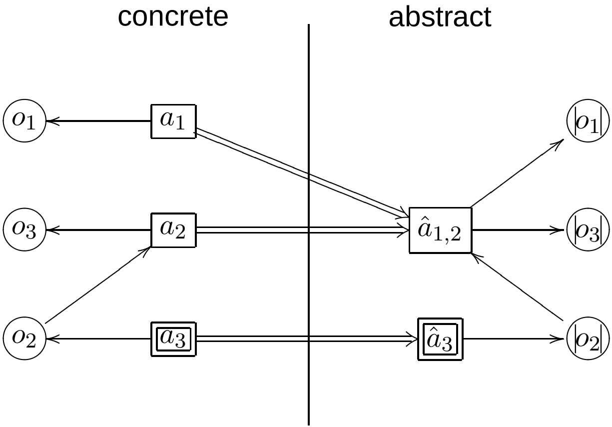
Example: Abstract garbage collection

Uh-oh! Zombie born. Concrete-abstract symmetry broken.



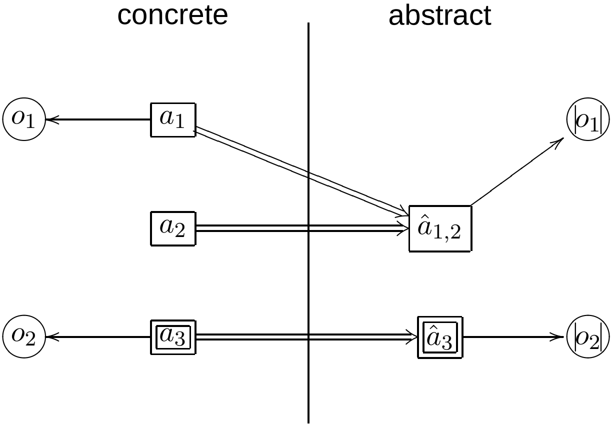
Example: Abstract garbage collection

Solution: Rewind and garbage collect first.



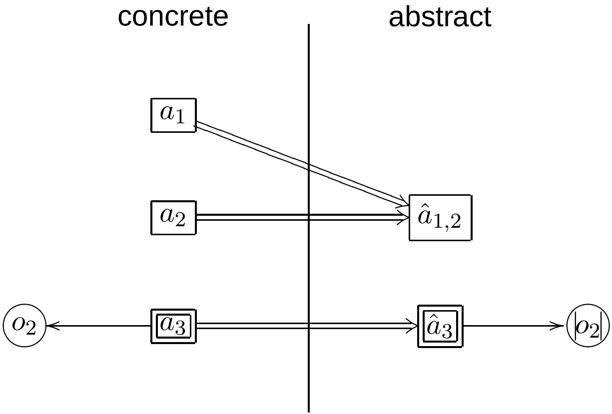
Example: Abstract garbage collection

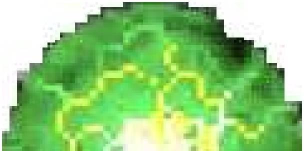
As it was:

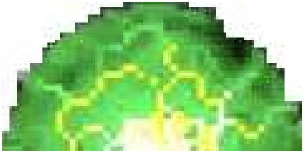


Example: Abstract garbage collection

After garbage collection:

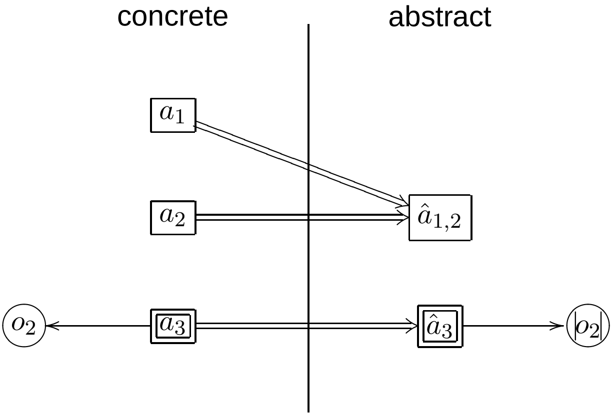






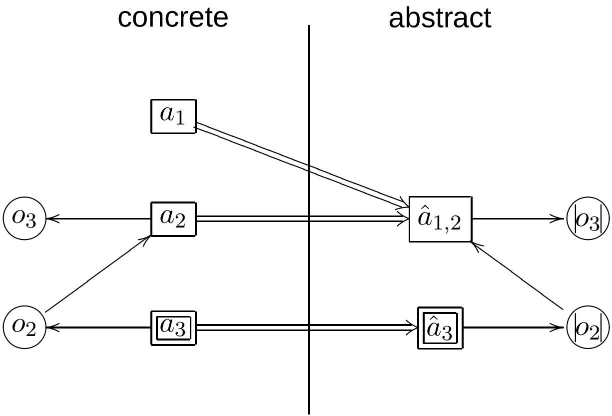
Example: Abstract garbage collection

Try again: Allocate object o_3 to address a_2. Point o_2 to a_2.



Example: Abstract garbage collection

No overapproximation!



Correctness of garbage collection

Theorem

Garbage collection does not change the meaning of a program:

$$\begin{tikzcd}
\varsigma_0 \arrow[r, Rightarrow] \arrow[d, leftrightarrow, "\equiv"'] & \varsigma_1 \arrow[r, Rightarrow] \arrow[d, leftrightarrow, "\equiv"'] & \varsigma_2 \arrow[r, Rightarrow] \arrow[d, leftrightarrow, "\equiv"'] & \varsigma_3 \arrow[r, Rightarrow] \arrow[d, leftrightarrow, "\equiv"'] & \varsigma_4 \arrow[r, Rightarrow] \arrow[d, leftrightarrow, "\equiv"'] & \cdots \\
\varsigma_0 \arrow[r, Rightarrow, "\Gamma"'] & \varsigma_1' \arrow[r, Rightarrow, "\Gamma"'] & \varsigma_2' \arrow[r, Rightarrow, "\Gamma"'] & \varsigma_3' \arrow[r, Rightarrow, "\Gamma"'] & \varsigma_4' \arrow[r, Rightarrow, "\Gamma"'] & \cdots
\end{tikzcd}$$

Soundness of the analysis

Theorem (Correctness of ΓCFA)

ΓCFA simulates the concrete semantics.

$$\begin{tikzcd}
\varsigma \arrow[r, "|\cdot|"] \arrow[d, "\Rightarrow"'] & |\varsigma| \arrow[r, "\sqsubseteq"] & \widehat{\varsigma} \arrow[d, dashed, "\approx\!\!>"] \\
\varsigma' \arrow[r, "|\cdot|"'] & |\varsigma'| \arrow[r, "\sqsubseteq"'] & \widehat{\varsigma'}
\end{tikzcd}$$

Abstract garbage collection & polyvariance

Question

Consider `(`λ `(...  k) ...)`. To where will it return?

0CFA

To everywhere called: Flow set for `k` grows monotonically.

ΓCFA with 0CFA contour set

To last call, if tail-recursive or leaf procedure.

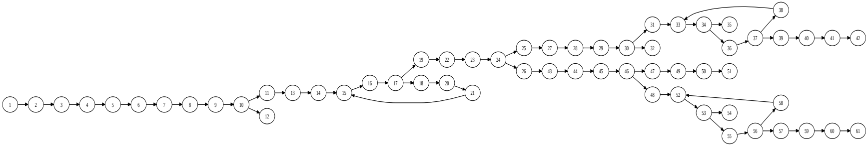
Example: Forking

```
(define (identity x) x)
(define mylock   (identity lock))
(define myunlock (identity unlock))
(mylock mutex)
(myunlock mutex)
```

Example: Forking

```
(define (identity x) x)
(define mylock   (identity lock))
(define myunlock (identity unlock))
(mylock mutex)
(myunlock mutex)
```

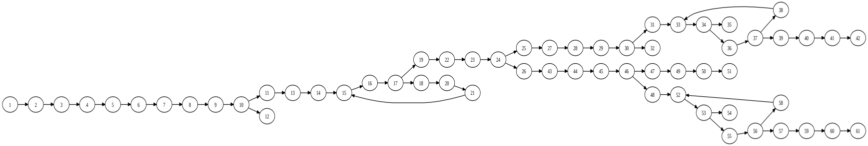
Without GC



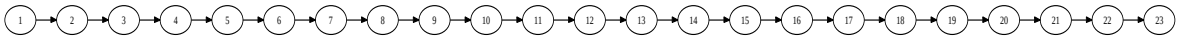
Example: Forking

```
(define (identity x) x)
(define mylock   (identity lock))
(define myunlock (identity unlock))
(mylock mutex)
(myunlock mutex)
```

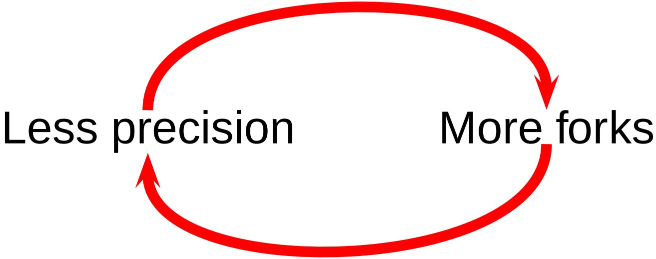
Without GC



With GC



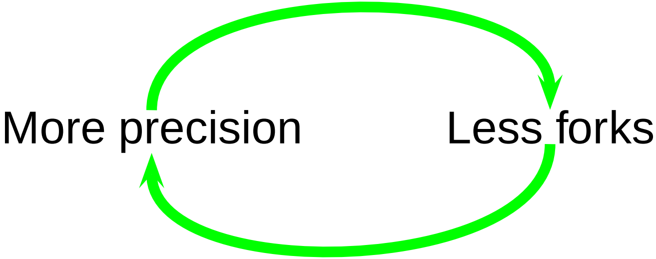
Vicious cycle



Less precision

More forks

Virtuous cycle

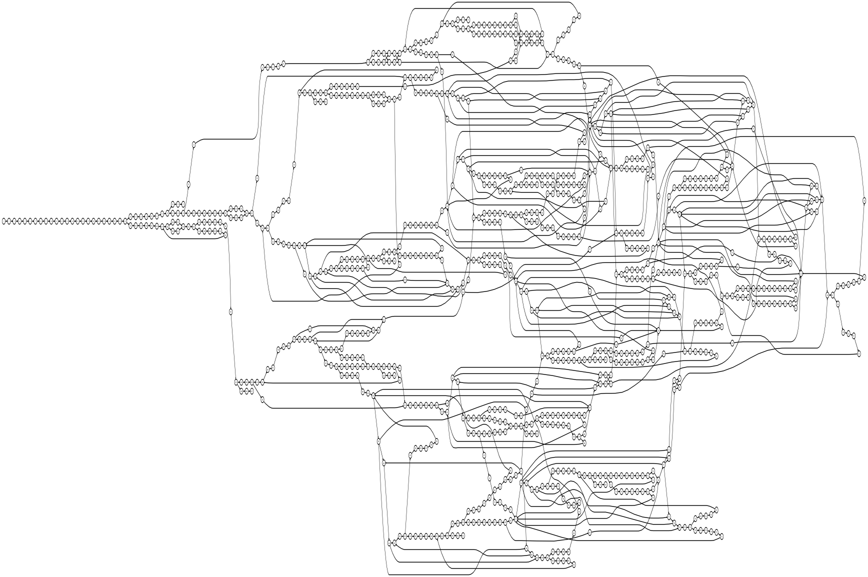


More precision

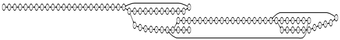
Less forks

Implementation & Results

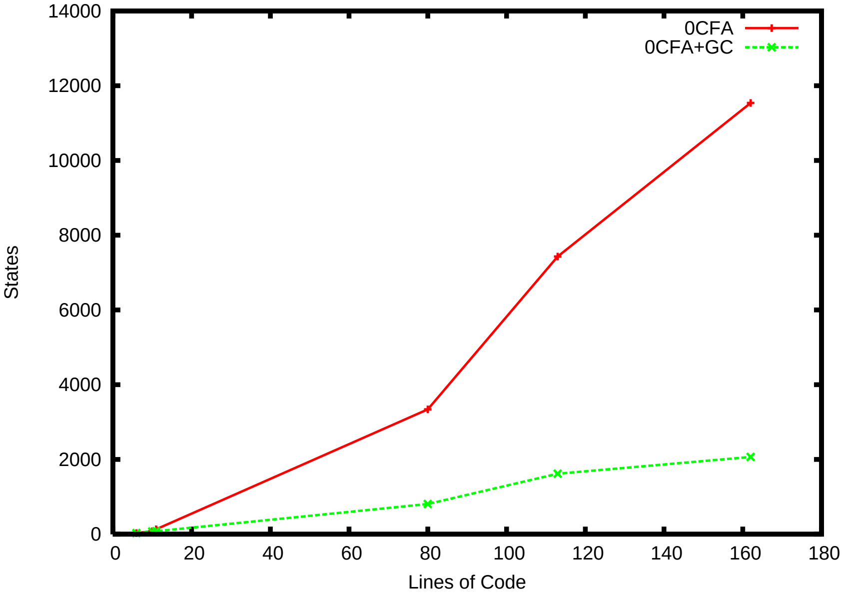
Without GC



With GC



Flow results: 0CFA & GC



Counting results: 0CFA & GC

program	% of variables with count ≤ 1
`earley`	94%
`int-fringe-coro`	89%
`int-stream-coro`	82%
`lattice`	91%
`nboyer`	98%
`perm`	95%
`put-double-coro`	92%
`sboyer`	98%

Results: 0CFA, GC, Counting & Widening

	$k = 0$,c,no-GC		$k = 0$,p		$k = 0$,c		$k = 0$,s	
`earley`	15%	258s	94%	24s	94%	15s	95%	90s
`int-fringe-coro`	26%	8s	87%	5s	87%	2s	89%	2s
`int-stream-coro`	14%	15s	79%	14s	79%	8s	82%	7s
`lattice`	12%	59s	91%	10s	91%	6s	OOM	$>$71m
`nboyer`	12%	68s	98%	93s	98%	48s	98%	18,420s
`perm`	8%	90s	95%	2s	95%	6s	95%	2s
`put-double-coro`	41%	2s	89%	2s	89%	1s	92%	0.8s
`sboyer`	OOM	$>$1,024s	98%	95s	98%	50s	OOM	$>$20,065s

- GC wins for precision & speed.
- Widening costs little precision.
- On average, widening saves time.

Results: 1CFA, GC, Counting & Widening

	$k = 1$,p		$k = 1$,c		$k = 1$,s	
`earley`	94%	143s	94%	83s	OOM	$>$45m
`int-fringe-coro`	88%	54s	88%	13s	92%	9s
`int-stream-coro`	87%	72s	87%	11s	90%	8s
`lattice`	91%	56s	92%	24s	OOM	$>$89m
`nboyer`	99%	221s	99%	231s	OOM	$>$164,040s
`perm`	95%	9s	95%	4s	95%	60s
`put-double-coro`	90%	12s	90%	4s	93%	2s
`sboyer`	98%	286s	OOM	$>$21,031s	OOM	$>$45,040s

- Widening costs little precision.
- On average, widening saves time.
- Small precision advantage to 1CFA.
- Large time cost to 1CFA.

Results: Improvements in super-β inlining

	0CFA+GC	
Program	Inlines w/o Counting	Inlines w/Counting
`fact-tail`	2	4
`fact-y-combinator`	4	8
`nested-loops`	4	10
`put-double-coroutines`	28	55
`integrate-fringe-coroutines`	45	77
`integrate-stream-coroutines`	46	72

Environment analysis, Take 2: $\triangle$CFA

Tool: Procedure strings

Classic model (Sharir & Pnueli, Harrison)

- Program trace at procedure level
- String of procedure activation/deactivation actions

Actions

control: call/return

stack: push/pop

Tool: Procedure strings

Classic model (Sharir & Pnueli, Harrison)

- Program trace at procedure level
- String of procedure activation/deactivation actions

Actions

control: call/return
stack: push/pop

`(fact 1)`

call `fact` / call `zero?` / return `zero?` / call – / return – /
call `fact` / call `zero?` / return `zero?` / return `fact` /
call $*$ / return $*$ / return `fact`

Note: Call/return items nest like parens.

CPS & stacks

But wait! CPS is all calls, no returns!

Procedure strings won’t nest properly:
call `a` / call `b` / call `c` / call `d` / ...

CPS & stacks

But wait! CPS is all calls, no returns!

Procedure strings won’t nest properly:

call `a` / call `b` / call `c` / call `d` / ...

Not necessarily.

User/continuation partition of CPS

Recursive factorial

```
(λ_t (n ktop)
  (letrec ((f (λ_f (m k)
                (%if0 m
                  (λ_1 () (k 1))
                  (λ_2 ()
                    (- m 1 (λ_3 (m2)
                             (f m2 (λ_4 (a)
                                     (* m a k)
                                     )))))))))
    (f n ktop)))
```

CPS conversion adds blue/red annotations,
permitting frame-push/frame-pop execution model.

Two control constructs

- Call a function.
- Call a continuation.

Problem

Meaning of call/return still a little murky.

Solution

Track stack behavior: push/pop.

Frame strings

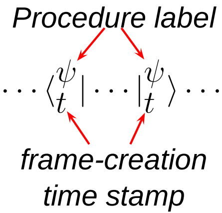
$\langle^{a}_{6}|\langle^{b}_{7}|^{b}_{7}\rangle|^{a}_{6}\rangle$

$|^{q}_{38}\rangle\langle^{q}_{38}|$

$\langle^{r}_{21}|^{r}_{21}\rangle\langle^{a}_{71}|$

$\langle^{a}_{4}|\langle^{b}_{5}|^{b}_{5}\rangle\langle^{c}_{6}|$

Anatomy of a frame-string character



Procedure label

$$\cdots\langle^{\psi}_{t}|\cdots|^{\psi}_{t}\rangle\cdots$$

frame-creation
time stamp

Modelling control/env with frame strings & CPS

A vocabulary for describing computational structure

Tail call (iteration): $|{}^{\gamma}_{\cdot}\rangle \cdots |{}^{\gamma}_{\cdot}\rangle|{}^{l}_{\cdot}\rangle\langle{}^{l}_{\cdot}|$

Non-tail call: $\langle{}^{l}_{\cdot}|$

Simple return: $|{}^{\gamma}_{\cdot}\rangle \cdots |{}^{\gamma}_{\cdot}\rangle|{}^{l}_{\cdot}\rangle\langle{}^{\gamma}_{\cdot}|$

Primop call: $\langle{}^{l}_{\cdot}||{}^{l}_{\cdot}\rangle\langle{}^{\gamma}_{\cdot}|$ or $|{}^{\gamma}_{\cdot}\rangle \cdots |{}^{\gamma}_{\cdot}\rangle|{}^{l}_{\cdot}\rangle\langle{}^{l}_{\cdot}||{}^{l}_{\cdot}\rangle\langle{}^{\gamma}_{\cdot}|$

“Upward” throw: $|{}^{\cdot}_{\cdot}\rangle \cdots |{}^{\cdot}_{\cdot}\rangle\langle{}^{\gamma}_{\cdot}|$

“Downward” throw: $|{}^{\cdot}_{\cdot}\rangle \cdots |{}^{\cdot}_{\cdot}\rangle\langle{}^{\cdot}_{\cdot}| \cdots \langle{}^{\cdot}_{\cdot}|\langle{}^{\gamma}_{\cdot}|$

Coroutine switch: $|{}^{\cdot}_{\cdot}\rangle \cdots |{}^{\cdot}_{\cdot}\rangle\langle{}^{\cdot}_{\cdot}| \cdots \langle{}^{\cdot}_{\cdot}|\langle{}^{\gamma}_{\cdot}|$

Handles *any* stack-to-stack delta.

Eval-state transition

$$\frac{}{(\llbracket \texttt{(}f\ e^*\ q^*\texttt{)}_\kappa \rrbracket, \beta, ve, \quad t) \Rightarrow (\mathit{proc}, \mathbf{d}, \mathbf{c}, ve, \quad t)}$$

$$\text{where } \begin{cases} \mathit{proc} = \mathcal{A}\,\beta\,ve\,t\,f \\ d_i = \mathcal{A}\,\beta\,ve\,t\,e_i \\ c_j = \mathcal{A}\,\beta\,ve\,t\,q_j \end{cases}$$

Apply-state transition

$$\frac{\mathit{length}(\mathbf{d}) = \mathit{length}(\mathbf{u}) \qquad \mathit{length}(\mathbf{c}) = \mathit{length}(\mathbf{k})}{((\llbracket \texttt{(}\lambda_\psi\ \texttt{(}u^*\ k^*\texttt{)}\ \mathit{call}\texttt{)} \rrbracket, \beta, t_b), \mathbf{d}, \mathbf{c}, ve, \quad t) \Rightarrow (\mathit{call}, \beta', ve', \quad t')}$$

$$\text{where } \begin{cases} t' = \mathit{tick}(t) \\ \beta' = \beta[u_i \mapsto t', k_j \mapsto t'] \\ ve' = ve[(u_i, t') \mapsto d_i, (k_j, t') \mapsto c_j] \end{cases}$$

Eval-state transition

$$\frac{}{\left(\llbracket (f \; e^* \; q^*)_\kappa \rrbracket, \beta, ve, \delta, t\right) \;\Rightarrow\; (proc, \mathbf{d}, \mathbf{c}, ve, \delta', t)}$$

$$\text{where } \begin{cases} proc = \mathcal{A}\, \beta\, ve\, t\, f \\ d_i = \mathcal{A}\, \beta\, ve\, t\, e_i \\ c_j = \mathcal{A}\, \beta\, ve\, t\, q_j \\ \nabla\varsigma = \begin{cases} (age_\delta\; proc)^{-1} & f \in CEXP \\ (youngest_\delta\; \mathbf{c})^{-1} & \text{otherwise} \end{cases} \\ \delta' = \delta + (\lambda t. \nabla\varsigma) \end{cases}$$

Apply-state transition

$$\frac{length(\mathbf{d}) = length(\mathbf{u}) \qquad length(\mathbf{c}) = length(\mathbf{k})}{((\llbracket (\lambda_\psi \; (u^* \; k^*) \; call) \rrbracket, \beta, t_b), \mathbf{d}, \mathbf{c}, ve, \delta, t) \;\Rightarrow\; (call, \beta', ve', \delta', t')}$$

$$\text{where } \begin{cases} t' = tick(t) \\ \beta' = \beta[u_i \mapsto t', k_j \mapsto t'] \\ ve' = ve[(u_i, t') \mapsto d_i, (k_j, t') \mapsto c_j] \\ \nabla\varsigma = \langle^{\psi}_{t'} | \\ \delta' = (\delta + (\lambda t. \nabla\varsigma))[t' \mapsto \epsilon] \end{cases}$$

Eval-state transition

$$\frac{}{(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \ \gtrapprox\ (\widehat{proc}, \widehat{\mathbf{d}}, \widehat{\mathbf{c}}, \widehat{ve}, \widehat{\delta}', \widehat{t})}$$

$$\text{where } \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ f \\ \widehat{d}_i = \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ e_i \\ \widehat{c}_i = \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ q_i \\ \Delta\widehat{p} = \begin{cases} \left(\widehat{age}_{\widehat{\delta}}\,\{\widehat{proc}\}\right)^{-1} & f \in \mathit{EXPC} \\ \left(\widehat{youngest}_{\widehat{\delta}}\,\widehat{\mathbf{c}}\right)^{-1} & \text{otherwise} \end{cases} \\ \widehat{\delta}' = \widehat{\delta} \oplus (\lambda\widehat{t}.\Delta\widehat{p}) \end{cases}$$

Apply-state transition

$$\frac{\mathit{length}(\widehat{\mathbf{d}}) = \mathit{length}(\mathbf{u}) \qquad \mathit{length}(\widehat{\mathbf{c}}) = \mathit{length}(\mathbf{k})}{((\llbracket (\lambda_\psi\ (u^*\ k^*)\ \mathit{call}) \rrbracket, \widehat{\beta}, \widehat{t}_b), \widehat{\mathbf{d}}, \widehat{\mathbf{c}}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \ \gtrapprox\ (\mathit{call}, \widehat{\beta}', \widehat{ve}', \widehat{\delta}', \widehat{t}')}$$

$$\text{where } \begin{cases} \widehat{t}' = \widehat{tick}(\widehat{t}) \\ \widehat{\beta}' = \widehat{\beta}[u_i \mapsto \widehat{t}', k_j \mapsto \widehat{t}'] \\ \widehat{ve}' = \widehat{ve}\ \sqcup\ \left[(u_i, \widehat{t}') \mapsto \widehat{d}_i, (k_j, \widehat{t}') \mapsto \widehat{c}_j\right] \\ \Delta\widehat{p} = |\langle^{\psi}_{\widehat{t}'}|| \\ \widehat{\delta}' = \left(\widehat{\delta} \oplus (\lambda\widehat{t}.\Delta\widehat{p})\right)\ \sqcup\ \left[\widehat{t}' \mapsto |\epsilon|\right] \end{cases}$$

Soundness theorem

Theorem (Analysis safety)

ΔCFA simulates the concrete semantics.

$$\begin{tikzcd}
\varsigma \arrow[r, "|\cdot|"] \arrow[d, "\Rightarrow"'] & {|\varsigma|} \arrow[r, "\sqsubseteq"] & \widehat{\varsigma} \arrow[d, dashed, "\approx\!\!>"] \\
\varsigma' \arrow[r, "|\cdot|"'] & {|\varsigma'|} \arrow[r, "\sqsubseteq"'] & \widehat{\varsigma}'
\end{tikzcd}$$

Connecting frame strings & environments

Interval notation for frame-string change

$$[t, t'] = \delta_{t'}(t)$$

Theorem

Environments separated by continuation frame actions differ by the continuations' bindings.

$$\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle \langle^{\gamma'_1}_{t_1}| \cdots \langle^{\gamma'_n}_{t_m}| \implies \beta_{t_1}|\overline{B(\boldsymbol{\gamma'})} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}.$$

(Note: inferring t_0/t_1 environment relationship from log at time t_2.)

Concrete super-β condition

In English

λ expression ψ may be inlined at call site κ if,
whenever we call a procedure from call site κ,

- it is a closure over ψ, and
- the closure environment and the call-site environment have identical bindings for the free vars of ψ.

As mathematics

$$\mathit{Inlinable}((\kappa, \psi), pr) = \forall(\llbracket(f\ e^*\ q^*)_\kappa\rrbracket, \beta, ve, \delta, t) \in \mathcal{V}(pr) :$$
$$\text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f$$
$$\text{then } \begin{cases} \psi = \psi' \\ \beta_b|\mathit{free}(L_{pr}(\psi)) = \beta|\mathit{free}(L_{pr}(\psi)) \end{cases}$$

Correctness theorem

Theorem (Super-β transform safety)

$Inlinable((\kappa, \psi), pr)$*-directed inlining does not change meaning of program.*

$$\begin{tikzcd}
\varsigma \arrow[r, "||\cdot||"] \arrow[d, "S"'] & ||\varsigma|| & S^{-1}\varsigma_S \arrow[l, "||\cdot||"'] \\
S\varsigma \arrow[r, "||\cdot||"'] & ||\varsigma_S|| & \varsigma_S \arrow[l, "||\cdot||"] \arrow[u, "S^{-1}"']
\end{tikzcd}$$

Concrete super-β conditions—in frame-string terms

$$
\begin{aligned}
&\mathit{Local\text{-}Inlinable}((\kappa,\psi),pr) = \forall(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\qquad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\qquad \text{then } \begin{cases} \psi = \psi' \\ \exists\boldsymbol{\gamma} : \begin{cases} \lfloor[t_b, t]\rfloor \succ^{\boldsymbol{\gamma}} \epsilon \\ \mathit{free}(L_{pr}(\psi)) \subseteq \overline{B(\boldsymbol{\gamma})}. \end{cases} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Escaping\text{-}Inlinable}((\kappa,\psi),pr) = \\
&\qquad \forall(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\qquad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\qquad\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi')) : \exists\boldsymbol{\gamma} : \begin{cases} \lfloor[\beta(v), t]\rfloor \succ^{\boldsymbol{\gamma}} \lfloor[t_b, t]\rfloor \\ v \notin B(\boldsymbol{\gamma}). \end{cases} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{General\text{-}Inlinable}((\kappa,\psi),pr) = \\
&\qquad \forall(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\qquad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\qquad\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi')) : \lfloor[\beta(v), t]\rfloor = \lfloor[\beta_b(v), t]\rfloor. \end{cases}
\end{aligned}
$$

Abstract super-β conditions—in frame-string terms

$$\widehat{Local\text{-}Inlinable}((\kappa,\psi),pr) = \forall(\llbracket(f\ e^*\ q^*)_\kappa\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) :$$

$$\text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f$$

$$\text{then } \begin{cases} \psi = \psi' \\ \exists\boldsymbol{\gamma} : \begin{cases} \widehat{\delta}(\widehat{t}_b) \succsim^{\boldsymbol{\gamma}} |\epsilon| \\ \mathit{free}(L_{pr}(\psi)) \subseteq \overline{B(\boldsymbol{\gamma})}. \end{cases} \end{cases}$$

$$\widehat{Escaping\text{-}Inlinable}((\kappa,\psi),pr) \iff$$

$$\forall(\llbracket(f\ e^*\ q^*)_\kappa\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) :$$

$$\text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f$$

$$\text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi')) : \exists\boldsymbol{\gamma} : \begin{cases} \widehat{\delta}(\widehat{\beta}(v)) \succsim^{\boldsymbol{\gamma}} \widehat{\delta}(\widehat{t}_b) \\ v \notin B(\boldsymbol{\gamma}). \end{cases} \end{cases}$$

$$\widehat{General\text{-}Inlinable}((\kappa,\psi),pr) =$$

$$\forall(\llbracket(f\ e^*\ q^*)_\kappa\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) :$$

$$\text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi'), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f$$

$$\text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi')) : \widehat{\delta}(\widehat{\beta}(v)) = \widehat{\delta}(\widehat{\beta}_b(v)). \end{cases}$$

Applications

Application: Globalization

Transformation

Turn `x` into global variable.

Condition

Measure of `x` never exceeds 1.

Payoff

Smaller (possibly eliminated) environments in closures.

Application: Register-allocated environments

Transformation

Allocate escaping variables to registers.

Condition

- Variables interfere if state exists where both have measure ≥ 1.
- Color interference graph with registers.

Payoff

Smaller environments, faster code.

Application: Super-β copy propagation

Transformation

Replace reference `x` with reference `z`.

Condition

In states using `x`, value bound to `x` $\equiv$ value bound to `z`.

Payoff

- May make `x` useless.
- Enables continuation promotion.
- Enables coroutine fusion.

Application: Static closure allocation

Transformation

Allocate environment record for closure at compile time.

Condition

Measure of λ term never exceeds 1.

Payoff

- Eliminates stack and heap allocation.
- Eliminates record offset computation.

Application: Super-β rematerialization

Transformation

Inline λ term where free variables not available.

Condition

Values of non-available free-variables are recomputable.

Payoff

Smaller (possibly eliminated) enviroment records for closures.

Application: Super-β teleportation

Transformation

Inline λ term where free variables not available.

Condition

Free variables can be moved to common scope, *e.g.*, globalized.

Payoff

Smaller (possibly eliminated) environment records for closures.

Application: Must-alias analysis

Condition

- Two abstract addresses are equal.
- Both have measure 1.

Payoff

- Strong update: better precision.
- Double-free detection.
- Use of freed memory detection.

Application: Escape analysis

Transformation

Turn heap allocation into stack allocation.

Condition

Net stack motion from creation to use is pushes.

Payoff

Cheaper allocation.

Application: Lightweight continuation conversion

Transformation

Convert continuations from stacks to stack pointers.

Condition

Net stack motion from creation to use is pushes.

Payoff

Cheaper continuations in common case.

Application: Static `setjmp/longjmp` verification

Condition

Net stack motion from creation to use is pushes.

Payoff

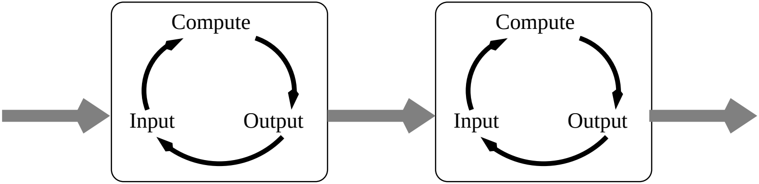
Program will never return to smashed stack.

Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only. . .

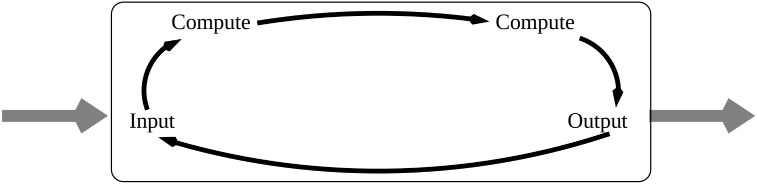


Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only. . .

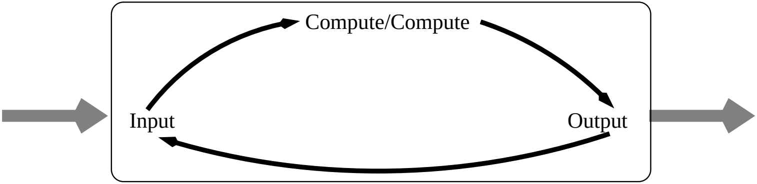


Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only…

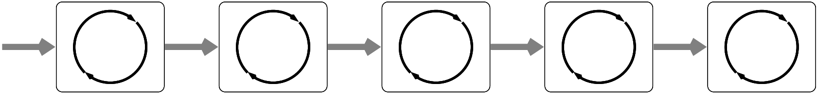


Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only. . .

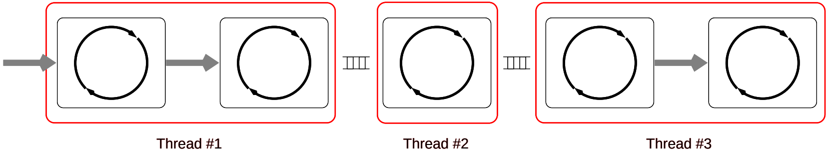


Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only. . .

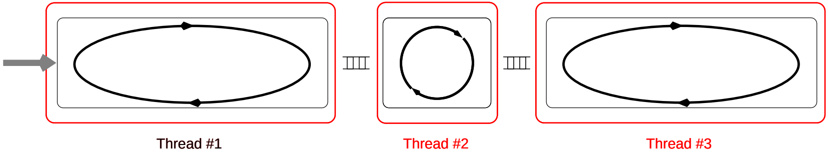


Application: Transducer/process fusion

Process Pipelines

- Unix pipes, *e.g.*, `find . | grep foo`
- Graphics pipelines.
- Network stacks.
- DSP networks.

If only…



Application: Logic-flow analysis

Mechanical (flow): $\hat{\varsigma} \longrightarrow \hat{\varsigma}' \longrightarrow \hat{\varsigma}'' \longrightarrow \cdots$

Propositional (logic): $\Pi \dashrightarrow \Pi' \dashrightarrow \Pi'' \dashrightarrow \cdots$

Application: Logic-flow analysis

$$\begin{tikzcd}
\text{Mechanical (flow):} & \hat{\varsigma} \arrow[r] \arrow[dr, dotted] & \hat{\varsigma}' \arrow[r] \arrow[dr, dotted] & \hat{\varsigma}'' \arrow[r] \arrow[dr, dotted] & \cdots \\
\text{Propositional (logic):} & \Pi \arrow[r, dotted] \arrow[ur] & \Pi' \arrow[r, dotted] \arrow[ur] & \Pi'' \arrow[r, dotted] \arrow[ur] & \cdots
\end{tikzcd}$$

Counting bootstraps propositions from mechanical interpretation.

Related work

Cousot^2, 1977:	Abstract interpretation.
Sestoft, 1988:	Globalization.
Shivers, 1988:	k-CFA.
Harrison, 1989:	Procedure strings.
Shivers, 1991:	Re-flow analysis.
Wand & Steckler, 1994:	Invariance-set analysis.
Jagannathan *et al.*, 1998:	Higher-order must-alias analysis.

Distinctions: Re-flow analysis

- k-CFA re-run for each contour of interest: *Expensive*.
- No abstract garbage collection.
- Assertion: Subsumed by μCFA.

Distinctions: Invariance-set analysis

- Specific kind of environment analysis: lexical *v*. dynamic.
- No abstract garbage collection.
- Constraint-based.
- Fixed context-sensitivity: 0CFA.
- Less general: Supports fewer applications.

Results: Γ+μCFA & Invariance-set Analysis

	$\theta^+\Gamma^+$	$\theta^+\Gamma^-$	$\theta^-\Gamma^+$	$\theta^-\Gamma^-$	Γ^+/θ^+	Time$_\theta$	Time$_\Gamma$
`earley`	61	0	649	239	1100%	14s+2s	24s
`int-fringe-coro`	136	0	24	25	117%	1s+ϵs	5s
`int-stream-coro`	129	0	4	36	103%	5s+ϵs	14s
`lattice`	79	0	70	40	200%	7s+ϵs	10s
`nboyer`	231	0	44	22	188%	43s+5s	68s
`perm`	140	0	149	17	206%	1s+ϵs	2s
`put-double-coro`	72	0	17	7	123%	ϵs+ϵs	2s
`sboyer`	235	0	50	22	121%	49s+5s	95s

- Invariance-set analysis faster.
- Counting & collection more precise.

Distinctions: Higher-order must-alias analysis

- Requires repeated runs of analysis.
- Uses *flat* lattice of cardinality.
- Constraint-based.
- Fixed widening: Per-point.
- Fixed context-sensitivity: 0CFA.
- Empirically subsumed by Γ+μCFA.
- Less general: Supports fewer applications.

Results: MAA & Γ+μCFA

	MAA[1]		$k = 0$,p		$k = 0$,c†		$k = 0$,s	
`earley`	15%	258s	94%	24s	94%	15s	95%	90s
`int-fringe-coro`	26%	8s	87%	5s	87%	2s	89%	2s
`int-stream-coro`	14%	15s	79%	14s	79%	8s	82%	7s
`lattice`	12%	59s	91%	10s	91%	6s	OOM	$>$71m
`nboyer`	12%	68s	98%	93s	98%	48s	98%	18,420s
`perm`	8%	90s	95%	2s	95%	6s	95%	2s
`put-double-coro`	41%	2s	89%	2s	89%	1s	92%	0.8s
`sboyer`	OOM	$>$1,024s	98%	95s	98%	50s	OOM	$>$20,065s

- Counting & collection faster.
- Counting & collection more precise.

† theoretical fixed point of iterated MAA.

Future & ongoing work

- Tighter, unaided coroutine fusion.
- Reformulations for OO (Java-Shimple), imperative (LLVM-SSA).
- Invariance-flow analysis (ΘCFA).
- Anodized contours.
- Garbage-collectible model of pointer arithmetic.
- Partial abstract GC for polyvariance.
- Lazy configuration-widening for multithreaded programs.
- PDA-based abstractions for abstract frame strings in ΔCFA.

Contributions

- Unified framework for general environment analysis.
- Two *independent* solutions to the environment problem:
  - One based on counting.
  - One based on frame strings.
- Proof of correctness for super-β inlining.
- Abstract GC: Enhanced precision via resource management.

Thank you.

How often do you garbage collect?

When zombie creation is imminent.
In practice, one in four transitions.

Accumulating propositions: Equality

Proposition

$$\forall x \in Conc_{\hat{\varsigma}}(\widehat{b}_1) : \forall y \in Conc_{\hat{\varsigma}}(\widehat{b}_2) : x = y.$$

Condition for inclusion

- Binding $\widehat{b}_1$ to $\widehat{b}_2$.
- Measure of both does not exceed 1.

Payoff

Boosts super-β rematerialization, copy propagation.

Accumulating propositions: Conditions

Proposition

c in `(if` c e_{true} e_{false}`)`.

Condition for inclusion

- Measure of c does not exceed 1.
- Assert c on fork to e_{true}.
- Assert not c on fork to e_{false}.

Payoff

Assists run-time check removal, verification.

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

$$\begin{aligned} \texttt{id} &\mapsto \\ \texttt{x} &\mapsto \\ \texttt{(id 3)} &\mapsto \\ \texttt{y} &\mapsto \\ \texttt{(id 4)} &\mapsto \end{aligned}$$

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

1. `(λ (x) x)` flows to `id`.

```
    id ↦ (λ (x) x)
     x ↦
(id 3) ↦
     y ↦
(id 4) ↦
```

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

1. `(λ (x) x)` flows to `id`.
2. Then, `3` flows to `x`.

```
    id ↦ (λ (x) x)
     x ↦ 3
(id 3) ↦
     y ↦
(id 4) ↦
```

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

1. `(λ (x) x)` flows to `id`.
2. Then, `3` flows to `x`.
3. Then, `3` flows to `y`, `(id 3)`; $\texttt{x} \mapsto 3$ now dead.

$$\begin{aligned} \texttt{id} &\mapsto \texttt{(λ (x) x)} \\ \texttt{x} &\mapsto \cancel{3} \\ \texttt{(id 3)} &\mapsto 3 \\ \texttt{y} &\mapsto 3 \\ \texttt{(id 4)} &\mapsto \end{aligned}$$

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

1. `(λ (x) x)` flows to `id`.
2. Then, `3` flows to `x`.
3. Then, `3` flows to `y`, `(id 3)`;
   `x` ↦ `3` now dead.
4. Then, `4` flows to `x`.

$\texttt{id} \mapsto \texttt{(}\lambda\texttt{ (x) x)}$

$\texttt{x} \mapsto \cancel{\texttt{3}}\ \texttt{4}$

$\texttt{(id 3)} \mapsto \texttt{3}$

$\texttt{y} \mapsto \texttt{3}$

$\texttt{(id 4)} \mapsto$

ΓCFA & Precision

```
(let* ((id (λ (x) x))
       (y (id 3)))
 (id 4))
```

ΓCFA thinks...

1. `(λ (x) x)` flows to `id`.
2. Then, `3` flows to `x`.
3. Then, `3` flows to `y`, `(id 3)`; `x` ↦ 3 now dead.
4. Then, `4` flows to `x`.
5. Then, `4` flows to `(id 4)`.

`id` ↦ `(λ (x) x)`
`x` ↦ ~~3~~ 4
`(id 3)` ↦ 3
`y` ↦ 3
`(id 4)` ↦ 4