Environment Analysis of Higher-Order Languages

Matthew Might

Georgia Institute of Technology

7 June 2007
Environment analysis is feasible and useful for higher-order languages.
Points

Environment analysis is **feasible**:  
- Abstract frame strings. (POPL 2006, TCS 2007)  
- Configuration-widening, *etc.*

Environment analysis is **useful**:  
- Super-\(\beta\) 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-$\beta$ inlining.
- Super-$\beta$ copy propagation.
- Static closure allocation.
- Super-$\beta$ rematerialization.
- Super-$\beta$ 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

- \( x \mapsto 3, \ y \mapsto 4 \)
- \( x \mapsto "foo" \)

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
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 $\lambda$

In one construct, $\lambda$ is:

- control,
- environment,
- and data.
Outline

- Develop $k$-CFA.
- Formalize environment problem.
- Build abstract counting.
- Make it feasible: abstract garbage collection.
- Tour $\Delta$CFA.
- Review applications.
- Look at related work.
What is $k$-CFA?

$k$-CFA
Where do $\lambda$ 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 $\lambda$ terms flow?

Example

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

(map (\ (x) (+ x 1)) '(1 2 3)) ; '(2 3 4)
(map (\ (x) (- x 1)) '(1 2 3)) ; '(0 1 2)
```
What is $k$-CFA?

$k$-CFA
Where do $\lambda$ terms flow?

Example

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

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

(map (\ (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

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

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

class Cat extends Animal { ... }

Animal fido = ... ;
fido.eat();
```
What about environments?

Closure = $\lambda$ term + environment
Object = class + struct

```
(define (f x) (\ (z) (+ x z)))

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

(loop 0)
```
What about environments?

Closure = $\lambda$ term + environment
Object = class + struct

\[
\text{(define (f x) (\lambda (z) (+ x z)))}
\]

\[
\text{(define (loop n)}
\text{\quad (display ((f n) n))}
\text{\quad (loop (+ n 2)))}
\]

\[
\text{(loop 0)}
\]
What about environments?

Closure = \( \lambda \) term + environment
Object = class + struct

\[
\text{(define (f x) (} \lambda (z) (+ x z)))
\]

\[
\text{(define (loop n)}
\]
\[
\text{(display ((f n) n))}
\]
\[
\text{(loop (+ n 2))}
\]

\[
\text{(loop 0)}
\]

\[
\begin{align*}
[x & \mapsto 0] \\
[x & \mapsto 2] \\
[x & \mapsto 4] \\
[x & \mapsto 6] \\
[x & \mapsto 8] \\
[x & \mapsto 10] \\
[x & \mapsto 12] \\
\ldots
\end{align*}
\]
What about environments?

Closure = \( \lambda \) term + environment
Object = class + struct

\[
\begin{align*}
(\text{define } (f \ x) (\lambda (z) (+ x z))) \\
(\text{define } (\text{loop } n)) \\
(\text{display } ((f \ n) \ n)) \\
(\text{loop } (+ n 2))
\end{align*}
\]

\[
\begin{align*}
[x \mapsto 0] \\
[x \mapsto 2] \\
[x \mapsto 4] \\
[x \mapsto 6] \\
[x \mapsto 8] \\
[x \mapsto 10] \\
[x \mapsto 12] \\
\ldots
\end{align*}
\]

Environments must merge during finite analysis.
What about environments?

Closure = $\lambda$ term + environment
Object = class + struct

```scheme
(define (f x) (\ (z) (+ x z)))
(define (loop n)
  (display ((f n) n))
  (loop (+ n 2)))
(loop 0)
```

$x \mapsto \text{int}$
Problem: Merging blocks reasoning

Unsound reasoning

$|x| = |y|$, but $x \neq y!$
Problem: Merging blocks reasoning

Concrete Space

Abstract Space

Unsound reasoning

\[|a| = |w| = |b| \text{ does imply } a = b.\]
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

(\texttt{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

\[(\lambda (\ ) x) + [x \mapsto 3]\]

\[
\begin{align*}
\text{(let ((f (\lambda (x h) (if (zero? x) h (\lambda ( ) x)))) (f 0 (f 3 #f)) )
\end{align*}
\]

Fact: \((\lambda ( ) x)\) flows to \((h)\).

Question: Safe to super-\(\beta\) inline?
Problem: Inability to judge environments

Fact: $(\lambda () x)$ flows to $(h)$.

Question: Safe to super-$\beta$ inline?
Problem: Inability to judge environments

\[(\lambda (x) x) \xrightarrow{} 3\]

\[
\begin{align*}
\text{(let } ((f \ (\lambda (x \ h) \ (\text{if} \ (\text{zero?} \ x) \ h) \ (\lambda () x))))
\end{align*}
\]

\[
\begin{align*}
(f \ 0 \ (f \ 3 \ #f)))
\end{align*}
\]

Fact: \((\lambda () x) \) flows to \((h)\).

Question: Safe to super-\(\beta\) inline?
Problem: Inability to judge environments

(\lambda ( x ) \mapsto 3)

(let ((f (\lambda (x h) (if (zero? x)
   (((\lambda () x))
    (\lambda () x)))))
  (f 0 (f 3 #f)))

Fact: (\lambda () x) flows to (h).

Question: Safe to super-\beta inline?
Problem: Inability to judge environments

\[(\lambda (\ ) x) + [x \mapsto 3]\]

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

Fact: \((\lambda (\ ) x)\) flows to \((h)\).

Question: Safe to super-\(\beta\) 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

$\text{Env} = \text{Variable} \rightarrow \text{Value}$
Tool: Factored environments in $k$-CFA

$Env = \text{Variable} \rightarrow \lambda \times Env$
Tool: Factored environments in $k$-CFA

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

Must break recursion for analysis.
Tool: Factored environments in $k$-CFA

$$Env = \text{Variable} \rightarrow \text{Time}$$
Tool: Factored environments in $k$-CFA

$Env = Variable \rightarrow Time$

$Variable \times Time \rightarrow Value$

$Binding$
Tool: Factored environments in $k$-CFA

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

$$\text{Variable} \times \text{Time} \rightarrow \text{Value}$$

Merge environments by partitioning Time into finite number of sets.
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: \( \varsigma_1 \)

Abstract: \( \hat{\varsigma}_1 \)
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: \[ s_1 \rightarrow s_2 \]

Abstract: \[ \hat{s}_1 \]
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: \( s_1 \rightarrow s_2 \rightarrow s_3 \)

Abstract: \( \widehat{s}_1 \)
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$

Abstract: $\hat{s}_1$
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: \( \varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3 \rightarrow \varphi_4 \rightarrow \varphi_5 \rightarrow \cdots \)

Abstract: \( \hat{\varphi}_1 \)
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: \[ s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow \cdots \]

Abstract: \[ \hat{s}_1 \rightarrow \hat{s}_2 \]
Tool: Abstract interpretation

Definition

**Abstract interpretation** approximates set of reachable states.

Interpretation

Concrete: \( \mathfrak{s}_1 \rightarrow \mathfrak{s}_2 \rightarrow \mathfrak{s}_3 \rightarrow \mathfrak{s}_4 \rightarrow \mathfrak{s}_5 \rightarrow \cdots \)

Abstract: \( \hat{\mathfrak{s}}_1 \rightarrow \hat{\mathfrak{s}}_2 \rightarrow \hat{\mathfrak{s}}_3 \)
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow \cdots$

Abstract: $\hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4$

$\hat{s}_{3.1,1}$

$\hat{s}_{3.2,1}$
Tool: Abstract interpretation

Definition

Abstract interpretation approximates set of reachable states.

Interpretation

Concrete: $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow \cdots$

Abstract: $\hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4$

$\hat{s}_{3,1,1} \rightarrow \hat{s}_{3,1,2}$

$\hat{s}_{3.2,1}$
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

\[
e, f \in EXP ::= v \\
| (\lambda (v_1 \cdots v_n) \text{call}) \\
call \in CALL ::= (f \; e_1 \cdots e_n)
\]
CPS narrows concern

\( \lambda \) is universal representation of control & env.

<table>
<thead>
<tr>
<th>Construct</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun call</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>fun return</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>iteration</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>sequencing</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>conditional</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>exception</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td>coroutine</td>
<td>call to ( \lambda )</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
</tbody>
</table>

**Advantage**

Now \( \lambda \) is fine-grained construct.
Strategy

- Define state machine: $\varsigma \Rightarrow \varsigma'$.
- $\kappa$-CFA = abstract interpretation of $\Rightarrow$. 
Semantics: Eval states

\[ (, , , ) \]
Semantics: Eval states

\[(\llbracket f \; e_1 \cdots e_n \rrbracket, , , )\]
Semantics: Eval states

\[ \langle \langle f \ e_1 \cdots e_n \rangle \rangle, \beta, \ \rangle \]

Call site  Var \rightarrow\ Time
Semantics: Eval states

$$\mathcal{A}(v, \beta, ve) = \begin{cases} 
\text{let } t_{\text{bound}} &= \beta(v) \\
\text{value} &= ve(v, t_{\text{bound}}) \\
\text{in } &\text{value}
\end{cases}$$
Semantics: Eval states

\[
\mathcal{A}(v, \beta, ve) = \begin{cases} 
\text{let } t_{\text{bound}} = \beta(v) \\
\text{value } = ve(v, t_{\text{bound}}) \\
\text{in } value
\end{cases}
\]
Semantics: Eval states

$\mathcal{A}(v, \beta, ve) = \begin{cases} 
\text{let } t_{\text{bound}} = \beta(v) \\
\text{value} = ve(v, t_{\text{bound}}) \\
\text{in } \text{value}
\end{cases}$
Semantics: Eval states

Procedure

\[ \text{proc} = A(f, \beta, ve) \]

\[ \left[ \left[ f\ e_1 \cdots e_n \right] \right], \beta, ve, t \Rightarrow (\text{proc}, , , , ) \]

\[ A(v, \beta, ve) = \begin{cases} \text{let } t_{\text{bound}} = \beta(v) \\ \text{value} = ve(v, t_{\text{bound}}) \end{cases} \]

\[ A(\text{lam}, \beta, ve) = (\text{lam}, \beta) \]
Semantics: Eval states

**Procedure Arguments**

\[
\begin{align*}
\text{proc} &= A(f, \beta, ve) \\
\text{\textit{d}}_i &= A(e_i, \beta, ve) \\
\text{\textit{d}}(\llbracket f e_1 \cdots e_n \rrbracket, \beta, ve, t) &\Rightarrow (\text{proc}, \text{\textit{d}}, \ldots)
\end{align*}
\]

\[
A(v, \beta, ve) = \begin{cases} 
\text{let } t_{\text{bound}} = \beta(v) \\
\text{value} = ve(v, t_{\text{bound}}) \\
\text{in } \text{value}
\end{cases}
\]

\[
A(\text{lam}, \beta, ve) = (\text{lam}, \beta)
\]
Semantics: Eval states

\[
\text{Procedure Arguments}
\]

\[
\begin{align*}
\text{proc} &= \mathcal{A}(f, \beta, ve) \\
\text{d}_i &= \mathcal{A}(e_i, \beta, ve) \\
(\llbracket f e_1 \cdots e_n \rrbracket, \beta, ve, t) \Rightarrow (\text{proc, d, ve, \_})
\end{align*}
\]

\[
\text{Var} \times \text{Time} \rightarrow \text{Val}
\]

\[
\mathcal{A}(v, \beta, ve) = \begin{cases} \\
\text{let} & t_{\text{bound}} = \beta(v) \\
\text{value} & = ve(v, t_{\text{bound}}) \\
\text{in} & \text{value} \\
\mathcal{A}(\text{lam}, \beta, ve) &= (\text{lam}, \beta)
\end{cases}
\]
Semantics: Eval states

Procedure Arguments

\[
\begin{align*}
\text{proc} &= \mathcal{A}(f, \beta, ve) \\
\quad d_i &= \mathcal{A}(e_i, \beta, ve) \\
(\llbracket (f \ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (\text{proc}, d, ve, t + 1)
\end{align*}
\]

Var × Time → Val Timestamp

\[
\mathcal{A}(v, \beta, ve) = \begin{cases} 
\text{let } t_{\text{bound}} = \beta(v) \\
\text{value} = ve(v, t_{\text{bound}}) \\
\text{in } \text{value}
\end{cases}
\]

\[
\mathcal{A}(\text{lam}, \beta, ve) = (\text{lam}, \beta)
\]
Semantics: Apply states

(, , , )
Semantics: Apply states

\(((\llbracket \lambda (v_1 \cdots v_n) \text{ call} \rrbracket), \beta'), d, ve, t)\)
Semantics: Apply states

\[
(\langle [\lambda (v_1 \cdots v_n) \text{ call}], \beta' \rangle, d, ve, t) \Rightarrow (\quad, \quad, \quad, )
\]
Semantics: Apply states

$\left( \left[ \left( \lambda \left( v_1 \cdots v_n \right) \text{call} \right) \right], \beta' \right), d, ve, t) \Rightarrow (\text{call}, \ , \ )$
Semantics: Apply states

$$(((\lambda (v_1 \cdots v_n) \ call), \beta'), d, ve, t) \Rightarrow (\ call, \beta'[v_i \mapsto t], \ )$$
Semantics: Apply states

\[
([\lambda (v_1 \cdots v_n) \, \text{call}], \beta'), \, \mathbf{d}, \, \mathbf{ve}, \, t) \Rightarrow (\text{call}, \, \beta'[v_i \mapsto t], \, \mathbf{ve}[\langle v_i, t \rangle \mapsto d_i], )
\]
Semantics: Apply states

\[
(((\lambda (v_1 \cdots v_n) \text{call}), \beta'), d, ve, t) \Rightarrow (\text{call}, \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d], t)
\]
Eval-state transition

\[
\begin{align*}
\text{proc} &= A(f, \beta, ve) \\
\text{d}_i &= A(e_i, \beta, ve) \\
(\llbracket (f \ e_1 \cdots e_n) \rrbracket, \beta, ve, t) &\Rightarrow (\text{proc}, \text{d}, ve, t + 1)
\end{align*}
\]

Apply-state transition

\[
\begin{align*}
\text{proc} &= (\llbracket (\lambda (v_1 \cdots v_n) \ \text{call}) \rrbracket, \beta') \\
(\text{proc}, \text{d}, ve, t) &\Rightarrow (\text{call, } \beta'[v_i \mapsto t], ve[(v_i, t) \mapsto d_i], t)
\end{align*}
\]

Domains

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

Lookup function

\[
\begin{align*}
\mathcal{A}(\text{lam}, \beta, ve) &= (\text{lam}, \beta) \\
\mathcal{A}(v, \beta, ve) &= ve(v, \beta(v))
\end{align*}
\]
Eval-state transition

\[ \widehat{\text{proc}} \in \hat{A}(f, \hat{\beta}, \hat{ve}) \quad \widehat{d_i} = \hat{A}(e_i, \hat{\beta}, \hat{ve}) \]

\[ ([f \; e_1 \cdots e_n], \hat{\beta}, \hat{ve}, t) \leadsto (\widehat{\text{proc}}, \widehat{d}, \hat{ve}, \text{succ}(t)) \]

Apply-state transition

\[ \widehat{\text{proc}} = ([\lambda (v_1 \cdots v_n) \; \text{call}], \hat{\beta}') \]

\[ (\widehat{\text{proc}}, \widehat{d}, \hat{ve}, t) \leadsto (\text{call}, \hat{\beta}'[v_i \mapsto t], \hat{ve} \sqcup [(v_i, t) \mapsto \widehat{d_i}], t) \]

Domains

\[ \hat{\varsigma} \in \widehat{\text{Eval}} = \text{CALL} \times \text{BEnv} \times \text{VEnv} \times \text{Time} \]

\[ + \text{Apply} = \text{Proc} \times \hat{D}^* \times \text{VEnv} \times \text{Time} \]

\[ \hat{\beta} \in \text{BEnv} = \text{VAR} \rightarrow \text{Time} \]

\[ \hat{ve} \in \text{VEnv} = \text{VAR} \times \text{Time} \rightarrow \hat{D} \]

\[ \widehat{\text{proc}} \in \text{Proc} = \text{Clo} + \{\text{halt}\} \]

\[ \widehat{clo} \in \widehat{\text{Clo}} = \text{LAM} \times \text{BEnv} \]

\[ \hat{d} \in \hat{D} = \mathcal{P}(\text{Proc}) \]

\[ \hat{t} \in \text{Time} = \text{finite set of times (contours)} \]

Lookup function

\[ \hat{A}(\text{lam}, \hat{\beta}, \hat{ve}) \]

\[ = \{ (\text{lam}, \hat{\beta}) \} \]

\[ \hat{A}(v, \hat{\beta}, \hat{ve}) \]

\[ = \hat{ve}(v, \hat{\beta}(v)) \]
Eval-state transition

\[
\overline{\text{proc}} \in \hat{A}(f, \hat{\beta}, \hat{\nu}e) \quad \hat{d}_i = \hat{A}(e_i, \hat{\beta}, \hat{\nu}e) \\
([[(f \ e_1 \cdots e_n)], \hat{\beta}, \hat{\nu}e, t) \Rightarrow (\overline{\text{proc}}, \hat{d}, \hat{\nu}e, \text{succ}(t))
\]

Apply-state transition

\[
\overline{\text{proc}} = (\llbracket (\lambda \ (v_1 \cdots v_n) \ call) \rrbracket, \hat{\beta}') \\
(\overline{\text{proc}}, \hat{d}, \hat{\nu}e, t) \Rightarrow (\text{call}, \hat{\beta}'[v_i \mapsto t], \hat{\nu}e \sqcup [(v_i, t) \mapsto \hat{d}_i], t)
\]

Domains

\[
\hat{\varsigma} \in \hat{\text{Eval}} = \text{CALL} \times \hat{\text{BEnv}} \times \hat{\text{VEnv}} \times \hat{\text{Time}} \\
+ \hat{\text{Apply}} = \hat{\text{Proc}} \times \hat{\text{D}}^* \times \hat{\text{VEnv}} \times \hat{\text{Time}} \\
\hat{\beta} \in \hat{\text{BEnv}} = \text{VAR} \rightarrow \hat{\text{Time}} \\
\hat{\nu}e \in \hat{\text{VEnv}} = \text{VAR} \times \hat{\text{Time}} \rightarrow \hat{\text{D}} \\
\overline{\text{proc}} \in \hat{\text{Proc}} = \text{Clo} + \{\text{halt}\} \\
\overline{\text{clo}} \in \hat{\text{Clo}} = \text{LAM} \times \hat{\text{BEnv}} \\
\hat{d} \in \hat{\text{D}} = \mathcal{P}(\hat{\text{Proc}}) \\
\hat{t} \in \hat{\text{Time}} = \text{finite set of times (contours)}
\]

Lookup function

\[
\hat{A}(\text{lam}, \hat{\beta}, \hat{\nu}e) = \{(\text{lam}, \hat{\beta})\} \\
\hat{A}(\nu, \hat{\beta}, \hat{\nu}e) = \hat{\nu}e(\nu, \hat{\beta}(\nu))
\]
Environment analysis, Take 1: $\mu$CFA
Environment problem refined

Input
Two abstract environments, $\hat{\beta}_1$ and $\hat{\beta}_2$. 
Environment problem refined

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

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

Eval

$$(\llbracket (f \ e_1 \cdots e_n) \rrbracket, \widehat{\beta}, \widehat{\text{ve}}, \widehat{t}) \Rightarrow (\widehat{\text{proc}}, \widehat{d}, \widehat{\text{ve}}, \text{succ}(\widehat{t}))$$

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

Apply

$$((\llbracket (\lambda \ (v_1 \cdots v_n) \ \text{call}) \rrbracket, \widehat{\beta_b}), \widehat{d}, \widehat{\text{ve}}, \widehat{t}) \Rightarrow (\text{call}, \widehat{\beta}', \widehat{\text{ve}}', \widehat{t})$$

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

Abstract binding counter, \( \hat{\mu} : \text{"Bindings"} \to \{0, 1, \infty\} \).

Eval

\[
\left[ (f \ e_1 \cdots e_n) \right], \hat{\beta}, \hat{\nu} e, \hat{\mu}, \hat{t} \Rightarrow (\hat{\text{proc}}, \hat{d}, \hat{\nu} e, \hat{\mu}, \text{succ}(\hat{t}))
\]

where

\[
\begin{align*}
\hat{\text{proc}} & \in \hat{\mathcal{A}}(f, \hat{\beta}, \hat{\nu} e) \\
\hat{d}_i & = \hat{\mathcal{A}}(e_i, \hat{\beta}, \hat{\nu} e)
\end{align*}
\]

Apply

\[
\left[ (\lambda (v_1 \cdots v_n) \ \text{call}) \right], \hat{\beta}_b, \hat{d}, \hat{\nu} e, \hat{\mu}, \hat{t} \Rightarrow (\text{call}, \hat{\beta}', \hat{\nu} e', \hat{\mu}', \hat{t})
\]

where

\[
\begin{align*}
\hat{\beta}' & = \hat{\beta}_b[v_i \mapsto \hat{t}] \\
\hat{\nu} e' & = \hat{\nu} e \sqcup [(v_i, \hat{t}) \mapsto \hat{d}_i] \\
\hat{\mu}' & = \hat{\mu} \oplus [(v_i, \hat{t}) \mapsto 1]
\end{align*}
\]
Basic Principle
If \( \{x\} = \{y\} \), then \( x = y \).

Theorem (Environment condition)
If \( \hat{\beta}_1(v) = \hat{\beta}_2(v) \),
and \( \hat{\mu}(v, \hat{\beta}_1(v)) = \hat{\mu}(v, \hat{\beta}_2(v)) = 1 \),
then \( \beta_1(v) = \beta_2(v) \).
\( \mu \text{CFA environment condition} \)

Basic Principle
If \( \{x\} = \{y\} \), then \( x = y \).

Theorem (Environment condition)
If \( \hat{\beta}_1(v) = \hat{\beta}_2(v) \),
and \( \hat{\mu}(v, \hat{\beta}_1(v)) = \hat{\mu}(v, \hat{\beta}_2(v)) = 1 \),
then \( \beta_1(v) = \beta_2(v) \),
where: \( (v, \beta_1(v)) \in \text{dom}(ve) \),
and \( |\beta_i| \subseteq \hat{\beta}_i \),
and \( |ve|^\mu \subseteq \hat{\mu} \).
\(\mu\)CFA environment condition

Basic Principle
If \(\{x\} = \{y\}\), then \(x = y\).

Theorem (Environment condition)
If \(\hat{\beta}_1(v) = \hat{\beta}_2(v)\), and \(\hat{\mu}(v, \hat{\beta}_1(v)) = \hat{\mu}(v, \hat{\beta}_2(v)) = 1\), then \(\beta_1(v) = \beta_2(v)\), where: \((v, \beta_1(v)) \in \text{dom}(ve)\), and \(|\beta_i| \subseteq \hat{\beta}_i\), and \(|ve|^\mu \subseteq \hat{\mu}\).

Problem
Most counts hit \(\infty\): almost every variable bound more than once!
Making it feasible: $\Gamma$CFA
Example: Abstract garbage collection

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

concrete  abstract

\[
\begin{align*}
&\text{o}_1 \quad \Box \text{a}_1 \\
&\Box \text{a}_2 \\
&\Box \text{a}_3 \\
&\Box \hat{\text{a}}_{1,2} \\
&\Box \hat{\text{a}}_3 \\
&\text{\hat{o}}_1
\end{align*}
\]
Example: Abstract garbage collection

3-address concrete heap. 2-address abstract counterpart.
Example: Abstract garbage collection

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

![Diagram of concrete and abstract heaps with arrows connecting nodes](image-url)
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

Example: Abstract garbage collection

Solution: Rewind and garbage collect first.
Example: Abstract garbage collection

As it was:
Example: Abstract garbage collection

After garbage collection:

concrete \hspace{1cm} abstract

\[ a_1 \rightarrow \hat{a}_{1,2} \]

\[ a_2 \rightarrow \hat{a}_{1,2} \]

\[ a_3 \rightarrow \hat{a}_3 \]

\[ o_2 \rightarrow a_3 \]

\[ o_2 \rightarrow \hat{a}_3 \]
Example: Abstract garbage collection

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

concrete

abstract

\[ \begin{array}{c}
\text{a}_1 \\
\text{a}_2 \\
\text{a}_3 \\
\end{array} \quad \begin{array}{c}
\hat{a}_{1,2} \\
\hat{a}_3 \\
\end{array} \]

\[ \begin{array}{c}
o_2 \\
o_2 \\
\end{array} \]
Example: Abstract garbage collection

No overapproximation!

concrete

abstract

$\hat{a}_{1,2}$

$a_1$

$a_2$

$a_3$

$\hat{a}_3$

$o_2$

$o_3$

$|o_3|$

$|o_2|$
Correctness of garbage collection

Theorem

*Garbage collection does not change the meaning of a program:*

\[
\begin{align*}
\gamma_0 & \rightarrow \rightarrow \gamma_1 & \rightarrow \rightarrow \gamma_2 & \rightarrow \rightarrow \gamma_3 & \rightarrow \rightarrow \gamma_4 & \rightarrow \rightarrow \cdots \\
\equiv & \downarrow & \equiv & \downarrow & \equiv & \downarrow & \equiv \\
\gamma_0 & \rightarrow \gamma_1' & \rightarrow \gamma_2' & \rightarrow \gamma_3' & \rightarrow \gamma_4' & \rightarrow \cdots
\end{align*}
\]
Soundness of the analysis

Theorem (Correctness of $\Gamma$CFA)

$\Gamma$CFA simulates the concrete semantics.

$\vdash \cdot \rightarrow |\cdot| \rightarrow \approx \rightarrow \approx \approx \approx \approx$

$\vdash |\cdot| \rightarrow |\cdot| \rightarrow \approx \rightarrow \approx \approx \approx \approx$

$\vdash |\cdot| \rightarrow |\cdot| \rightarrow \approx \rightarrow \approx \approx \approx \approx$
Abstract garbage collection & polyvariance

Question
Consider ($\lambda (\ldots \, k) \ldots$). To where will it return?

$0CFA$
To everywhere called: Flow set for $k$ grows monotonically.

$\Gamma CFA$ with $0CFA$ contour set
To last call, if tail-recursive or leaf procedure.
(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

![Graph showing the relationship between lines of code and states for 0CFA and 0CFA+GC. The graph illustrates how the states increase with an increase in lines of code. The red line represents 0CFA, while the green dashed line represents 0CFA+GC.](image-url)
## Counting results: 0CFA & GC

<table>
<thead>
<tr>
<th>program</th>
<th>% of variables with count $\leq 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>earley</td>
<td>94%</td>
</tr>
<tr>
<td>int-fringe-coro</td>
<td>89%</td>
</tr>
<tr>
<td>int-stream-coro</td>
<td>82%</td>
</tr>
<tr>
<td>lattice</td>
<td>91%</td>
</tr>
<tr>
<td>nboyer</td>
<td>98%</td>
</tr>
<tr>
<td>perm</td>
<td>95%</td>
</tr>
<tr>
<td>put-double-coro</td>
<td>92%</td>
</tr>
<tr>
<td>sboyer</td>
<td>98%</td>
</tr>
</tbody>
</table>
Results: 0CFA, GC, Counting & Widening

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

- GC wins for precision & speed.
- Widening costs little precision.
- On average, widening saves time.
Results: 1CFA, GC, Counting & Widening

<table>
<thead>
<tr>
<th>Method</th>
<th>$k = 1,p$</th>
<th>$k = 1,c$</th>
<th>$k = 1,s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>earley</td>
<td>94% 143s</td>
<td>94% 83s</td>
<td>OOM &gt;45m</td>
</tr>
<tr>
<td>int-fringe-coro</td>
<td>88% 54s</td>
<td>88% 13s</td>
<td>92% 9s</td>
</tr>
<tr>
<td>int-stream-coro</td>
<td>87% 72s</td>
<td>87% 11s</td>
<td>90% 8s</td>
</tr>
<tr>
<td>lattice</td>
<td>91% 56s</td>
<td>92% 24s</td>
<td>OOM &gt;89m</td>
</tr>
<tr>
<td>nbboyer</td>
<td>99% 221s</td>
<td>99% 231s</td>
<td>OOM &gt;164,040s</td>
</tr>
<tr>
<td>perm</td>
<td>95% 9s</td>
<td>95% 4s</td>
<td>95% 60s</td>
</tr>
<tr>
<td>put-double-coro</td>
<td>90% 12s</td>
<td>90% 4s</td>
<td>93% 2s</td>
</tr>
<tr>
<td>sboyer</td>
<td>98% 286s</td>
<td>OOM &gt;21,031s</td>
<td>OOM &gt;45,040s</td>
</tr>
</tbody>
</table>

- Widening costs little precision.
- On average, widening saves time.
- Small precision advantage to 1CFA.
- Large time cost to 1CFA.
## Results: Improvements in super-$\beta$ inlining

<table>
<thead>
<tr>
<th>Program</th>
<th>0CFA+GC</th>
<th>0CFA+GC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inlines w/o Counting</td>
<td>Inlines w/ Counting</td>
</tr>
<tr>
<td>fact-tail</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>fact-y-combinator</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>nested-loops</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>put-double-coroutines</td>
<td>28</td>
<td>55</td>
</tr>
<tr>
<td>integrate-fringe-coroutines</td>
<td>45</td>
<td>77</td>
</tr>
<tr>
<td>integrate-stream-coroutines</td>
<td>46</td>
<td>72</td>
</tr>
</tbody>
</table>
Environment analysis, Take 2: $\Delta$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? / return fact /
call * / return * / return fact

Note: Call/return items nest like parens.
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:
  \texttt{call a / call b / call c / call d / \ldots}

Not necessarily.
User/continuation partition of CPS

Recursive factorial

$$\lambda_t (n \ ktop)
\text{(letrec ((f (\lambda_f (m \ k))
\%if0 m
  (\lambda_1 () (k 1))
  (\lambda_2 ()
    (- m 1 (\lambda_3 (m2)
      (f m2 (\lambda_4 (a)
        (* m a k)
        ))))))))
(f n \ ktop)))$$

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

\[ \langle a_6 | \langle b_7 | b_7 \rangle | a_6 \rangle \]
\[ |q_{38} \rangle \langle q_{38} | \]
\[ \langle r_{21} | r_{21} \rangle \langle a_{71} | \]
\[ \langle a_4 | b_5 \langle b_5 \rangle | c_6 \rangle \]

Anatomy of a frame-string character

\[ \cdots \langle \psi | \cdots | \psi \rangle \cdots \]

Procedure label

frame-creation

time stamp
Modelling control/env with frame strings & CPS

A vocabulary for describing computational structure

Tail call (iteration): $|\gamma\rangle \cdots |\gamma\rangle |l\rangle \langle l|$

Non-tail call: $\langle l|$  

Simple return: $|\gamma\rangle \cdots |\gamma\rangle |l\rangle \langle \gamma|$

Primop call: $\langle l| |l\rangle \langle \gamma| \text{ or } |\gamma\rangle \cdots |\gamma\rangle |l\rangle \langle l||l\rangle \langle \gamma|$

“Upward” throw: $|:\rangle \cdots |:\rangle \langle \gamma|$  

“Downward” throw: $|:\rangle \cdots |:\rangle \langle :| \cdots \langle :| \langle \gamma|$  

Coroutine switch: $|:\rangle \cdots |:\rangle \langle :| \cdots \langle :| \langle \gamma|$

Handles any stack-to-stack delta.
Eval-state transition

\[
[(f e^* q^*)_\kappa], \beta, ve, \ t \Rightarrow (\text{proc}, d, c, ve, \ t)
\]

\[
\begin{align*}
\text{proc} &= A \beta ve t f \\
\text{d}_i &= A \beta ve t e_i \\
\text{c}_j &= A \beta ve t q_j
\end{align*}
\]

where

Apply-state transition

\[
\text{length}(d) = \text{length}(u) \quad \text{length}(c) = \text{length}(k)
\]

\[
((\Lambda_{\psi} (u^* k^*) \text{call})), \beta, t_b, d, c, ve, \ t \Rightarrow (\text{call}, \beta', ve', \ t')
\]

\[
\begin{align*}
t' &= \text{tick}(t) \\
\beta' &= \beta[\text{u}_i \mapsto t', \text{k}_j \mapsto t'] \\
\text{ve}' &= \text{ve}[(\text{u}_i, t') \mapsto \text{d}_i, (\text{k}_j, t') \mapsto \text{c}_j]
\end{align*}
\]

where
Eval-state transition

\[
\left[ (f \ e^* q^*) \kappa \right], \beta, ve, \delta, t \Rightarrow (\text{proc}, d, c, ve, \delta', t)
\]

where

\[
\begin{align*}
\text{proc} &= A \beta ve t f \\
\text{d}_i &= A \beta ve t e_i \\
\text{c}_j &= A \beta ve t q_j
\end{align*}
\]

\[
\nabla \varsigma = \begin{cases} 
\frac{\text{age}_\delta \text{proc}}{-1} & f \in CEXP \\
\frac{\text{youngest}_\delta c}{-1} & \text{otherwise}
\end{cases}
\]

\[
\delta' = \delta + (\lambda t. \nabla \varsigma)
\]

Apply-state transition

\[
\text{length}(d) = \text{length}(u) \quad \text{length}(c) = \text{length}(k)
\]

\[
\left[ (\Lambda_{\psi} (u^* k^*) \text{call}) \right], \beta, t_b, d, c, ve, \delta, t \Rightarrow (\text{call}, \beta', ve', \delta', t')
\]

where

\[
\begin{align*}
\text{t}' &= \text{tick}(t) \\
\beta' &= \beta[u_i \mapsto t', k_j \mapsto t'] \\
ve' &= ve[(u_i, t') \mapsto d_i, (k_j, t') \mapsto c_j]
\end{align*}
\]

\[
\nabla \varsigma = \langle \psi |_{t'} \rangle \\
\delta' = (\delta + (\lambda t. \nabla \varsigma))[t' \mapsto \epsilon]
\]
Eval-state transition

\[(\left[ \left( f \ u^* \ k^* \right) \right]_{\kappa}, \beta, \widehat{ve}, \widehat{\delta}, \hat{t}) \approx (\widehat{\text{proc}}, \widehat{d}, \widehat{c}, \widehat{ve}, \widehat{\delta}', \hat{t})\]

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

Apply-state transition

\[\text{length}(\widehat{d}) = \text{length}(u) \quad \text{length}(\widehat{c}) = \text{length}(k)\]

\[(\left[ \left( \Lambda_\psi (u^* \ k^*) \ \text{call} \right) \right], \beta, \widehat{t}_b, \widehat{d}, \widehat{c}, \widehat{ve}, \widehat{\delta}, \hat{t}) \approx (\text{call}, \beta', \widehat{ve}', \widehat{\delta}', \hat{t}')\]

\[
\begin{align*}
\text{where} & \quad \begin{cases} 
\hat{t}' = \text{tick}(\hat{t}) \\
\beta' = \beta[\hat{t}' \rightarrow t'] \\
\widehat{ve}' = \widehat{ve} \sqcup [(u_i, \hat{t}') \rightarrow \widehat{d}_i, (k_j, \hat{t}') \rightarrow \widehat{c}_j] \\
\Delta \widehat{p} = ||(u_{\hat{t}'})|| \\
\widehat{\delta}' = (\delta \oplus (\lambda \hat{t}. \Delta \widehat{p})) \sqcup [\hat{t}' \leftrightarrow |\epsilon|]
\end{cases}
\end{align*}
\]
Soundness theorem

Theorem (Analysis safety)

$\Delta \text{CFA simulates the concrete semantics.}$

$\begin{align*}
\tau & \xrightarrow{|\cdot|} |\tau| \xrightarrow{\subseteq} \hat{\tau} \\
\tau' & \xrightarrow{|\cdot|} |\tau'| \xrightarrow{\subseteq} \hat{\tau}'
\end{align*}$
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.

\[
[t_0, t_2] + [t_1, t_2]^{-1} = |\gamma_1^{i_1}\rangle \cdots |\gamma_n^{i_n}\rangle \langle t_1^{j_1}| \cdots \langle t_m^{j_m}| \implies \beta_{t_1} |B(\gamma')\rangle = \beta_{t_0} |\overline{B(\gamma)}\rangle.
\]

(Note: inferring \(t_0/t_1\) environment relationship from log at time \(t_2\).)
Concrete super-$\beta$ condition

In English

$\lambda$ expression $\psi$ may be inlined at call site $\kappa$ if, whenever we call a procedure from call site $\kappa$,

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

As mathematics

$$\text{Inlinable}((\kappa, \psi), pr) = \forall ([\left[ (f \ e^* \ q^*)_{\kappa} \right]], \beta, ve, \delta, t) \in \mathcal{V}(pr) :$$

if $\kappa = \kappa'$ and $(L_{pr}(\psi'), \beta_b, t_b) = A \beta ve t f$

then

$$\begin{cases} 
\psi = \psi' \\
\beta_b | free(L_{pr}(\psi)) = \beta | free(L_{pr}(\psi)) 
\end{cases}$$
Correctness theorem

**Theorem (Super-β transform safety)**

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

\[
\begin{align*}
\mathcal{S} \xrightarrow{||\cdot||} ||\mathcal{S}|| & \xleftarrow{||\cdot||} S^{-1}\mathcal{S}
\end{align*}
\]
Concrete super-$\beta$ conditions—in frame-string terms

Local-Inlinable((\(\kappa, \psi\), pr)) = \(\forall (\[ (f \ e^* \ q^*)_{\kappa} \], \beta, ve, \delta, t) \in V(pr) : \)

if \(\kappa = \kappa'\) and \((L_{pr}(\psi'), \beta_b, t_b) = A \beta ve t f\)

then \(\left\{\begin{array}{l}
\psi = \psi' \\
\exists \gamma : \{[[t_b, t]] \succ \gamma \in free(L_{pr}(\psi)) \subseteq \underbar{B(\gamma)}\}
\end{array}\right.\)

Escaping-Inlinable((\(\kappa, \psi\), pr)) =

\(\forall (\[ (f \ e^* \ q^*)_{\kappa} \], \beta, ve, \delta, t) \in V(pr) : \)

if \(\kappa = \kappa'\) and \((L_{pr}(\psi'), \beta_b, t_b) = A \beta ve t f\)

then \(\left\{\begin{array}{l}
\psi = \psi' \\
\forall v \in free(L_{pr}(\psi')) : \exists \gamma : \{[[\beta(v), t]] \succ \gamma \} \subseteq \underbar{B(\gamma)}\}
\end{array}\right.\)

General-Inlinable((\(\kappa, \psi\), pr)) =

\(\forall (\[ (f \ e^* \ q^*)_{\kappa} \], \beta, ve, \delta, t) \in V(pr) : \)

if \(\kappa = \kappa'\) and \((L_{pr}(\psi'), \beta_b, t_b) = A \beta ve t f\)

then \(\left\{\begin{array}{l}
\psi = \psi' \\
\forall v \in free(L_{pr}(\psi')) : [[\beta(v), t]] = [[\beta_b(v), t]].
\end{array}\right.\)
Abstract super-$\beta$ conditions—in frame-string terms

$\text{Local-Inlinable}((\kappa, \psi), pr) = \forall ([[(f \: e^* \: q^*)_\kappa]], \hat{\beta}, \hat{\nu} e, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}(pr):$

if $\kappa = \kappa'$ and $(L_{pr}(\psi'), \hat{\beta}_b, \hat{t}_b) = \hat{A}\hat{\beta} \hat{\nu} e \hat{t} f$

then
\[
\begin{cases}
\psi = \psi' \\
\exists \gamma : \left\{ \begin{array}{l}
\hat{\delta}(\hat{t}_b) \succcurlyeq \gamma |e| \\
\text{free}(L_{pr}(\psi)) \subseteq \overline{B(\gamma)}.
\end{array} \right.
\end{cases}
\]

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

\[
\forall ([[(f \: e^* \: q^*)_\kappa]], \hat{\beta}, \hat{\nu} e, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}(pr):$

if $\kappa = \kappa'$ and $(L_{pr}(\psi'), \hat{\beta}_b, \hat{t}_b) = \hat{A}\hat{\beta} \hat{\nu} e \hat{t} f$

then
\[
\begin{cases}
\forall v \in \text{free}(L_{pr}(\psi')) : \exists \gamma : \left\{ \begin{array}{l}
\hat{\delta}(\hat{\beta}(v)) \succcurlyeq \gamma \hat{\delta}(\hat{t}_b) \\
v \notin B(\gamma).
\end{array} \right.
\end{cases}
\]

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

\[
\forall ([[(f \: e^* \: q^*)_\kappa]], \hat{\beta}, \hat{\nu} e, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}(pr):$

if $\kappa = \kappa'$ and $(L_{pr}(\psi'), \hat{\beta}_b, \hat{t}_b) = \hat{A}\hat{\beta} \hat{\nu} e \hat{t} f$

then
\[
\begin{cases}
\psi = \psi' \\
\forall v \in \text{free}(L_{pr}(\psi')) : \hat{\delta}(\hat{\beta}(v)) = \hat{\delta}(\hat{\beta}_b(v)).
\end{cases}
\]
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 $\geq 1$.
- Color interference graph with registers.

Payoff
Smaller environments, faster code.
Application: Super-$\beta$ 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 $\lambda$ term never exceeds 1.

Payoff
- Eliminates stack and heap allocation.
- Eliminates record offset computation.
Application: Super-$\beta$ rematerialization

Transformation
Inline $\lambda$ term where free variables not available.

Condition
Values of non-available free-variables are recomputable.

Payoff
Smaller (possibly eliminated) environment records for closures.
Application: Super-$\beta$ teleportation

Transformation
Inline $\lambda$ 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, \textit{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):

\[ \hat{\varsigma} \rightarrow \hat{\varsigma}' \rightarrow \hat{\varsigma}'' \rightarrow \cdots \]

Propositional (logic):

\[ \Pi \rightarrow \Pi' \rightarrow \Pi'' \rightarrow \cdots \]
Application: Logic-flow analysis

Mechanical (flow):
\[ \hat{\varsigma} \rightarrow \hat{\varsigma}' \rightarrow \hat{\varsigma}'' \rightarrow \cdots \]

Propositional (logic):
\[ \Pi \rightarrow \Pi' \rightarrow \Pi'' \rightarrow \cdots \]

Counting bootstraps propositions from mechanical interpretation.
Related work


Shivers, 1988: $k$-CFA.


Distinctions: Re-flow analysis

- \( k \)-CFA re-run for each contour of interest: \textit{Expensive}.
- No abstract garbage collection.
- Assertion: Subsumed by \( \mu \text{CFA} \).
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: $\Gamma+\mu$CFA & Invariance-set Analysis

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

- 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 \( \Gamma + \mu \) CFA.
- Less general: Supports fewer applications.
Results: MAA & $Γ+\mu$CFA

<table>
<thead>
<tr>
<th></th>
<th>MAA$^\dagger$</th>
<th>$k = 0,p$</th>
<th>$k = 0,c^\dagger$</th>
<th>$k = 0,s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>earley</td>
<td>15%</td>
<td>258s</td>
<td>94%   24s</td>
<td>94% 15s</td>
</tr>
<tr>
<td>int-fringe-coro</td>
<td>26%</td>
<td>8s</td>
<td>87%   5s</td>
<td>87% 2s</td>
</tr>
<tr>
<td>int-stream-coro</td>
<td>14%</td>
<td>15s</td>
<td>79%   14s</td>
<td>79% 8s</td>
</tr>
<tr>
<td>lattice</td>
<td>12%</td>
<td>59s</td>
<td>91%   10s</td>
<td>91% 6s</td>
</tr>
<tr>
<td>nboyer</td>
<td>12%</td>
<td>68s</td>
<td>98%   93s</td>
<td>98% 48s</td>
</tr>
<tr>
<td>perm</td>
<td>8%</td>
<td>90s</td>
<td>95%   2s</td>
<td>95% 6s</td>
</tr>
<tr>
<td>put-double-coro</td>
<td>41%</td>
<td>2s</td>
<td>89%   2s</td>
<td>89% 1s</td>
</tr>
<tr>
<td>sboyer</td>
<td>OOM $&gt;1,024s$</td>
<td>98%</td>
<td>95s   50s</td>
<td>OOM $&gt;20,065s$</td>
</tr>
</tbody>
</table>

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

$^\dagger$ theoretical fixed point of iterated MAA.
Future & ongoing work

- Tighter, unaided coroutine fusion.
- Reformulations for OO (Java-Shimple), imperative (LLVM-SSA).
- Invariance-flow analysis ($\Theta$CFA).
- 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 $\Delta$CFA.
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-$\beta$ 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

\[ \forall x \in Conc_\xi(\hat{b}_1) : \forall y \in Conc_\xi(\hat{b}_2) : x = y. \]

Condition for inclusion

- Binding \( \hat{b}_1 \) to \( \hat{b}_2 \).
- Measure of both does not exceed 1.

Payoff

Boosts super-\( \beta \) rematerialization, copy propagation.
Accumulating propositions: Conditions

Proposition
\[ c \text{ in } (\text{if } c e_{\text{true}} e_{\text{false}}). \]

Condition for inclusion

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

Payoff

Assists run-time check removal, verification.
ΓCFA & Precision

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

ΓCFA thinks...

id ↦
    x ↦
    (id 3) ↦
    y ↦
    (id 4) ↦
(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 (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);
   x ↦→ 3 now dead.

id ↦→ (λ (x) x)
   x ↦→ Ø
(id 3) ↦→ 3
   y ↦→ 3
(id 4) ↦→
(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.
(let* ((id (λ (x) x))
       (y (id 3)))
  (id 4))

ΓCFA thinks...

1. \(\lambda (x) x\) flows to id.
   \[id \mapsto (\lambda (x) x)\]
   \[x \mapsto \emptyset\]
   \[4\]

2. Then, 3 flows to x.
   \[x \mapsto 3\]
   \[x \mapsto \emptyset\]
   \[4\]

3. Then, 3 flows to y, (id 3);
   \[x \mapsto 3\] now dead.
   \[(id 3) \mapsto 3\]
   \[y \mapsto 3\]

4. Then, 4 flows to x.
   \[(id 4) \mapsto 4\]

5. Then, 4 flows to (id 4).