Resolving and Exploiting the $k$-CFA Paradox

Matthew Might, Yannis Smaragdakis, and David Van Horn
Plan

★ What is $k$-CFA?
★ The paradox of $k$-CFA
★ A resolution: OO vs. functional $k$-CFA
★ An exploitation: $m$-CFA
★ Evaluation and conclusion
What is control flow analysis?

Answers the question: where does control transfer to at a call?

\[ o.m(5); \]

*Clearly a kind of points-to analysis.*
What is control flow analysis?

Answers the question: where does control transfer to at a call?

```java
n(o) { return o.m(5); }
```
What is control flow analysis?

Answers the question: where does control transfer to at a call?

\[
n(o) \{ \text{return } o.m(5); \} \\
\ldots \\
\text{if } (P) \ n(q) \ \text{else } n(r); 
\]
What is control flow analysis?

Answers the question: where does control transfer to at a call?

(5)
What is control flow analysis?

Answers the question: where does control transfer to at a call?

$$(\lambda (\mathcal{f}) (\mathcal{f} 5))$$
What is control flow analysis?

Answers the question: where does control transfer to at a call?

\[
\left( \lambda f \left( f \ 5 \right) \right) \\
\left( \text{if } P \ q \ r \right)
\]
What is control flow analysis?

Answers the question: where does control transfer to at a call?

But this is not easy to decide in a higher-order language.

Which languages are higher-order? Almost all of them: Scheme, ML, Haskell, JavaScript, Java, C, C++, Ruby, Python, etc.
A Simple use of Closures

; Number → (Number → Number)
(define (make-adder n)
  (λ (x)
    (+ x n)))

(define add5 (make-adder 5))
(define add3 (make-adder 3))

(add5 2) ;=> 7
(add3 7) ;=> 10

(make-adder 5) = ⟨(λ (x) (+ x n)), [n ↦ 5]⟩.
A Simple use of Closures

class MkAddr {
    makeAdder(n) {
        return new Add {
            add(x) { return x+n; }}}}

Non-idiomatic. Anonymous inner class emulates λ.
A Simple *non*-use of Closures

class Add {
    n;
    Add(n) { this.n = n; }
    add(x) { return x+n; }
}

“Closures” emulated by local fields and constructor accepting value of “free” variables.
What is $\kappa$-CFA?
Approximation in 0CFA

0CFA approximates closures by their code component.

$$(\text{make-adder } 5) = \langle (\lambda (x) (+ x n)), [n \mapsto 5] \rangle$$

$$(\lambda (x) (+ x n))$$

$$n \approx \{5, 3\}$$

$$(\text{add5 } 2) \approx \{7, 5\}$$

Context-insensitive points-to analysis.
What is $k$-CFA?

$k$-CFA uses *calling contexts* to increase analysis precision:

\[
\text{(make-adder 5)}^1 \ldots \text{(make-adder 3)}^2
\]

1-CFA:

\[
\ast \left\langle \left( \lambda \, (x) \, ( + \, x \, n ) \right), [n \mapsto 5] \right\rangle.
\]

\[
\ast \left\langle \left( \lambda \, (x) \, ( + \, x \, n ) \right), [n \mapsto 3] \right\rangle.
\]

\[
\text{(add5 2)} \approx \{ 7 \}
\]

*Call-site context-sensitive points-to analysis.*
The Paradox
**$k$-CFA is hard**

It did not take long to discover that the basic analysis, for any $k > 0$, was intractably slow for large programs.

*Shivers, Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned (2004)*
What makes $k$-CFA hard?

Closures.

$$(\lambda (f) (f \ 0) (f \ 1))$$

$$(\lambda (x) (\lambda (z) (z \ x)))$$

1-CFA:

$\star \langle (\lambda (z) (z \ x)) , [x \mapsto 0] \rangle.$  \hspace{1cm} 2 closures.

$\star \langle (\lambda (z) (z \ x)) , [x \mapsto 1] \rangle.$
What makes $k$-CFA hard?

$$\lambda (z) (z \ x_0 \ldots \ x_n)$$

1-CFA:

* $[x_1 \mapsto 0, \ldots, x_n \mapsto 0]$

* $[x_1 \mapsto 1, \ldots, x_n \mapsto 0]$

* ... 

* $[x_1 \mapsto 0, \ldots, x_n \mapsto 1]$

* $[x_1 \mapsto 1, \ldots, x_n \mapsto 1]$

$O(2^n)$ closures.

$k$-CFA is complete for EXPTIME (Van Horn and Mairson, ’08).
\( \kappa \)-CFA is easy

\( \kappa \)-CFA of object-oriented programs is in PTIME.

E.g., written in Datalog (Bravenboer and Smaragdakis, ’09).
$k$-CFA is easy

$k$-CFA of object-oriented programs is in PTIME.

E.g., written in Datalog (Bravenboer and Smaragdakis, ’09).

PTIME $\subsetneq$ EXPTIME
The Resolution
What makes (OO) $k$-CFA easy?

No closures.
Explicit closures: 1 variable

```java
class Add {
    n;
    Add(n) { this.n = n; }
    add(x) { return x+n; }
}

new Add(5);
new Add(2);
new Add(7);

Creates 3 “closures” over 1 variable.
```
Explicit closures: $C$ variables

class Add {
    n_0; \ldots n_c;
    Add(n_0, \ldots, n_c) { this.n_i = n_i; }
    add(x) { return x \ldots n_0 \ldots n_c; }
}

new Add(p_1, \ldots, p_c);

new Add(q_1, \ldots, q_c);

new Add(r_1, \ldots, r_c);

Creates 3 “closures” over $c$ variables.

Only consider 3 “closures”, not $3^c$!
Analytic message

“Explicit” closing collapses the value set.
Resolving the Paradox

$k$-CFA of:

★ an FP language with closures is EXPTIME.
★ an OO language without closures is PTIME.
Resolving the Paradox

$k$-CFA of:

- an FP language with closures is EXPTIME.
- an OO language without closures is PTIME.
- an OO language with closures is EXPTIME.
- an FP language without closures is PTIME.
Resolving the Paradox

$k$-CFA of:

- an FP language with closures is EXPTIME.
- an OO language without closures is PTIME.
- an OO language \textit{with} closures is EXPTIME.
- an FP language \textit{without} closures is PTIME.

But the $k$-CFA is \textit{the same}. 
The Exploitation
The idea: Flat closures

Change the representation of closures:

\[ \text{Clo} = \text{Lam} \times (\text{Var} \rightarrow \text{Addr}) \]

\[ \text{Clo}' = \text{Lam} \times \text{Addr} \]

At call sites, copy values of closure into the store.

*Essentially lifting \( \lambda \)-lifting into semantics.*
Evaluation and conclusion
## Speed: Worst-case benchmark

<table>
<thead>
<tr>
<th>Terms</th>
<th>$k = 1$</th>
<th>$m = 1$</th>
<th>poly.,$k=1$</th>
<th>$k=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>123</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>231</td>
<td>46 s</td>
<td>$\epsilon$</td>
<td>2 s</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>447</td>
<td>$\infty$</td>
<td>3 s</td>
<td>5 s</td>
<td>2 s</td>
</tr>
<tr>
<td>879</td>
<td>$\infty$</td>
<td>48 s</td>
<td>1 m 8 s</td>
<td>15 s</td>
</tr>
<tr>
<td>1743</td>
<td>$\infty$</td>
<td>51 m</td>
<td>$\infty$</td>
<td>3 m 48 s</td>
</tr>
</tbody>
</table>
## Speed and precision

<table>
<thead>
<tr>
<th>Prog/Terms</th>
<th>$k = 1$</th>
<th>$m = 1$</th>
<th>poly., $k=1$</th>
<th>$k=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>eta / 49</td>
<td>$\epsilon$ 7</td>
<td>$\epsilon$ 7</td>
<td>$\epsilon$ 3</td>
<td>$\epsilon$ 3</td>
</tr>
<tr>
<td>map / 157</td>
<td>$\epsilon$ 8</td>
<td>$\epsilon$ 8</td>
<td>$\epsilon$ 8</td>
<td>$\epsilon$ 6</td>
</tr>
<tr>
<td>sat / 223</td>
<td>$\infty$ -</td>
<td>$\epsilon$ 12</td>
<td>1s 12</td>
<td>$\epsilon$ 12</td>
</tr>
<tr>
<td>regex / 1015</td>
<td>4s 25</td>
<td>3s 25</td>
<td>14s 25</td>
<td>2s 25</td>
</tr>
<tr>
<td>scm2java / 2318</td>
<td>5s 86</td>
<td>3s 86</td>
<td>3s 79</td>
<td>4s 79</td>
</tr>
<tr>
<td>interp / 4289</td>
<td>5s 123</td>
<td>4s 123</td>
<td>9s 123</td>
<td>5s 123</td>
</tr>
<tr>
<td>scm2c / 6219</td>
<td>179s 136</td>
<td>143s 136</td>
<td>157s 131</td>
<td>55s 131</td>
</tr>
</tbody>
</table>

Column 1: analysis time, Column 2: inlinings enabled
Benchmark message

$m$-CFA v. $k$-CFA: as precise, but faster.
**Doggie bag**

- \( k \)-CFA of functional programs is hard because of closures.
- \( k \)-CFA of OO-programs is easy because of no closures.
- \( m \)-CFA: similar precision, faster.
- \( m \)-CFA is always in PTIME.
The End

Thank you.

Implementation and benchmarks:

http://www.ucombinator.org/projects/mcfa/