CFA2: Pushdown Flow Analysis for Higher-Order Languages

Dimitris Vardoulakis

Northeastern University
Flow analysis is instrumental in building good software.
Overview

Finite-state analyses and their limitations

CFA2 by example

Applications to JavaScript

Open problems
Finite-state analyses

Program as a graph whose nodes are the program points.
⇒ executions are strings in a regular language.
⇒ approximate program with finite-state machine.
⇒ call/return mismatch.
Finite-state analyses

Program as a graph whose nodes are the program points.
⇒ executions are strings in a regular language.
⇒ approximate program with finite-state machine.
⇒ call/return mismatch.

Fine for conditionals and loops (think Fortran).
Finite-state analyses

Program as a graph whose nodes are the program points.
⇒ executions are strings in a regular language.
⇒ approximate program with finite-state machine.
⇒ call/return mismatch.

Fine for conditionals and loops (think Fortran).

Call/return is the fundamental control-flow mechanism in HOLs. Finite-state analyses, such as \( k \)-CFA, have several limitations.
(define app (λ (f e) (f e)))
(define id (λ (x) x))

(let* ((n1 (app id 1))
        (n2 (app id 2)))
  (+ n1 n2))
0CFA example

main()

app id 1

n1

app id 2

n2

ret := n1+n2

main

app(f e)

f e

ret

app

id(x)

ret := x

id
0CFA example

1 main()

Global environment:
0CFA example

Global environment:
0CFA example

Global environment:

f  id
e  1
0CFA example

Global environment:

\[
\begin{array}{cc}
\text{f} & \text{id} \\
\text{e} & 1
\end{array}
\]
0CFA example

Global environment:

\[\begin{array}{ll}
  f & \text{id} \\
  e & 1 \\
  x & 1 \\
\end{array}\]
0CFA example

Global environment:

- f
- id
- e
- x
- ret-id

```
ret := n1 + n2
```

```
ret := x
```

```
app(f e)
```

```
app id 1
```
OCFA example

Global environment:

f  id
e  1
x  1
ret-id  1
0CFA example

Global environment:

- `f` : 1
- `id` : 1
- `e` : 1
- `ret-app` : 1
- `x` : 1
- `ret-id` : 1
0CFA example

Global environment:

- f
- id
- e
- ret-app
- x
- ret-id
OCFA example

Global environment:
- n1: 1
- f: id
- e: 1
- ret-app: 1
- x: 1
- ret-id: 1
OCFA example

Global environment:
- \( n1 \): 1
- \( f \): id
- \( e \): 1
- \( ret \): app 1
- \( ret \): id
- \( x \): 1
- \( ret \): id

```
main()

app id 1
n1

app id 2
n2

app(f e)
fe
ret
11

id(x)
ret := x
12

app
13

id
14
```
0CFA example

Global environment:

- `n1` with value 1
- `f` with value `id`
- `e` with values 1 and 2
- `ret-app` with values 1 and 2
- `x` with values 1 and 2
- `ret-id` with values 1 and 2
0CFA example

Global environment:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>id</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ret-app</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ret-id</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
OCFA example

Global environment:

\begin{align*}
\text{n1} & : 1 \ 2 \\
\text{n2} & : 1 \ 2 \\
\text{ret-main} & : 2 \ 3 \ 4 \\
\text{f} & : \text{id} \\
\text{e} & : 1 \ 2 \\
\text{ret-app} & : 1 \ 2 \\
\text{x} & : 1 \ 2 \\
\text{ret-id} & : 1 \ 2 \\
\end{align*}
0CFA example

Global environment:

<table>
<thead>
<tr>
<th></th>
<th>n1</th>
<th>n2</th>
<th>ret-main</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2 3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ret-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
0CFA example

Global environment:
- n1: 1 2
- n2: 1 2
- ret-main: 2 3 4
- f: id
- e: 1 2
- ret-app: 1 2
- x: 1 2
- ret-id: 1 2

Call/return mismatch causes spurious flow of data
⇒ commonly called functions pollute the analysis.
OCFA example

Global environment:

- **n1**: 1  2
- **n2**: 1  2

**ret-main**: 2  3  4

**f**: id
- **e**: 1  2
- **ret-app**: 1  2

**x**: 1  2
- **ret-id**: 1  2
0CFA example

Global environment:

- n1: 1 2
- n2: 1 2
- ret-main: 2 3 4
- f: id
- e: 1 2
- ret-app: 1 2
- x: 1 2
- ret-id: 1 2
0CFA example

Global environment:

- n1: 1 2
- n2: 1 2
- ret-main: 2 3 4
- f: id
- e: 1 2
- ret-app: 1 2
- x: 1 2
- ret-id: 1 2
Call/return mismatch causes spurious control flow
⇒ cannot accurately calculate stack change.

Global environment:

<table>
<thead>
<tr>
<th></th>
<th>n1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret-main</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret-app</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret-id</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Fake rebinding

(define (compose-same f x)
  (f (f x)))
Fake rebinding

\[
(\text{define (compose\textunderscore same} f x)
(f (f x)))
\]

\[
\lambda_a \quad \lambda_b
\]
Fake rebinding

\[
\text{(define (compose-same f x)}
\]
\[
\text{(f (f x)))}
\]

Flows:
\[
(f (f x))
\]
Fake rebinding

\[
\text{(define (compose-same } f x) \\
(f (f x)))
\]

Flows:
\[
(f (\lambda_a x))
\]
Fake rebinding

(\lambda a \ (\lambda a \ x))

Flows:
(\lambda a \ (\lambda a \ x))  ✓
Fake rebinding

\[
\text{(define (compose\text{-}same} f x) \\
(f (f x)))
\]

Flows:
\[
(\lambda_a (\lambda_a x)) \quad \checkmark \\
(\lambda_b (\lambda_b x)) \quad \checkmark
\]
Fake rebinding

\[
\text{(define (compose-same f x)}
\]
\[
(f (f x)))
\]

Flows:
\[
\begin{align*}
(\lambda_a (\lambda_a x)) & \quad \checkmark \\
(\lambda_b (\lambda_b x)) & \quad \checkmark \\
(\lambda_b (\lambda_a x)) & \quad \times
\end{align*}
\]
Fake rebinding

(define (compose-same f x)
  (f (f x)))

Flows:
(\lambda_a (\lambda_a x))  √
(\lambda_b (\lambda_b x))  √
(\lambda_b (\lambda_a x))  ✗
(\lambda_a (\lambda_b x))  ✗
(\lambda_a (\lambda_b x))  ✗
Imprecision slows down the analysis
Imprecision slows down the analysis

Imprecision

→

Spurious flows
to be analyzed
Imprecision slows down the analysis

Imprecision

Spurious flows to be analyzed

Flow data along spurious flows
Imprecision slows down the analysis

Imprecision -> Spurious flows to be analyzed

Spurious flows -> Flow data along spurious flows
The root cause: call/return mismatch

Causes spurious data flow.

Causes spurious control flow.

Leads to imprecision which slows down the analysis.

Fake rebinding?
CFA2 in a nutshell

Approximate a program as a PDA.
Use the stack for return-point information.
Unbounded call/return matching.
CFA2 in a nutshell

Approximate a program as a PDA.
Use the stack for return-point information.
Unbounded call/return matching.

A pushdown flow analysis [Sharir–Pnueli 81, Reps et al. 95].
CFA2 in a nutshell

Approximate a program as a PDA.
Use the stack for return-point information.
Unbounded call/return matching.

A pushdown flow analysis [Sharir–Pnueli 81, Reps et al. 95].

First-class functions, tail calls.
CFA2 in a nutshell

Approximate a program as a PDA. 
Use the stack for return-point information. 
Unbounded call/return matching.

A pushdown flow analysis [Sharir–Pnueli 81, Reps et al. 95].

First-class functions, tail calls.

Recursion causes stacks of unbounded size
⇒ infinite state space.
What we hope to achieve

Advanced reasoning about stack and environment:

- escape analysis for stack allocation
- super-$\beta$ inlining
- transducer fusion
What we hope to achieve

Advanced reasoning about stack and environment:

- escape analysis for stack allocation
- super-\(\beta\) inlining
- transducer fusion

Do old things better.

0CFA too imprecise.
Polyvariance didn’t help \(k\)-CFA much
and slowed it down a lot [Van Horn–Mairson 08].
Variable binding in CFA2

Binding environments:
- heap (like $k$-CFA)
- stack
Variable binding in CFA2

Binding environments:

- heap (like $k$-CFA)
- stack

Stack references: $(\lambda(x) (\lambda(y) (y\ (y\ x))))$
Bound in the top frame.
Stack references of same variable bound in same environment.
Variable binding in CFA2

Binding environments:
- heap (like $k$-CFA)
- stack

Stack references: $(\lambda(x) (\lambda(y) (y (y\ x))))$
Bound in the top frame.
Stack references of same variable bound in same environment.

Heap references: $(\lambda(x) (\lambda(y) (y (y\ x))))$
Either deeper in stack or in heap.
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))
(define merger
  (\lambda_1 (x) (\lambda_2 () x)))

(merger (\lambda_3 (y) y))

(define id
  (merger (\lambda_4 (z) z))())

(define comp-same
  (\lambda_5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger
x
id
comp-same
n1
n2

Stack:
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))
CFA2: pushdown automaton

(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

**Heap:**
merger  λ₁
x
id
comp-same
n1
n2

**Stack:**
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger
x
id
comp-same
n1
n2

Stack:

| x → λ₃ |
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger λ₁
 x λ₃
 id
comp-same
n1
n2

Stack:
CFA2: pushdown automaton

(define merger
  (λ1 (x) (λ2 () x)))

(merger (λ3 (y) y))

(define id
  (merger (λ4 (z) z))())

(define comp-same
  (λ5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger
x
id
comp-same
n1
n2

Stack:
  x → λ4
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z)))()

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger  merger  λ₁
x  x  λ₁
id  id  λ₃, λ₄
comp-same  comp-same
n1  n2

Stack:
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger x id comp-same n1 n2

Stack:
CFA2: pushdown automaton

(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger          λ₁
x               λ₃, λ₄
id              λ₃, λ₄
comp-same
n1               n2

Stack:
CFA2: pushdown automaton

(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n₁
  (comp-same id 1))

(define n₂
  (comp-same id 2))
CFA2: pushdown automaton

(define merger
  (lambda (x) (lambda () x)))

(merger (lambda y y))

(define id
  (merger (lambda z z)))

(define comp-same
  (lambda (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger    lambda_1
x          lambda_3, lambda_4
id         lambda_3, lambda_4
comp-same  lambda_5
n1
n2

Stack:
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z)))()

(define comp-same
  (λ₅ (f w) (f (f w)))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger  lambda_1
x lambda_3, lambda_4
id lambda_3, lambda_4
comp-same lambda_5
n1
n2

Stack:
| f ↦ {lambda_3, lambda_4}, w ↦ 1 |
(define merger
  (λ1 (x) (λ2 () x)))

(merger (λ3 (y) y))

(define id
  (merger (λ4 (z) z)))()

(define comp-same
  (λ5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger     λ₁
x          λ₃, λ₄
id         λ₃, λ₄
comp-same  λ₅
n1
n2

Stack:
| f ↦ λ₃, w ↦ 1 |
(define merger
  (\lambda_1 (x) (\lambda_2 () x)))

(merger (\lambda_3 (y) y))

(define id
  (merger (\lambda_4 (z) z))())

(define comp-same
  (\lambda_5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:

merger  \lambda_1
x    \lambda_3, \lambda_4
id  \lambda_3, \lambda_4
comp-same  \lambda_5
n1
n2

Stack:

\[
\begin{array}{c}
y \mapsto 1 \\
f \mapsto \lambda_3, w \mapsto 1
\end{array}
\]
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z)))()()

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger λ₁
x λ₃, λ₄
id λ₃, λ₄
comp-same λ₅
n1 n2

Stack:
\[
\begin{array}{c}
f \mapsto \lambda₃, w \mapsto 1
\end{array}
\]
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z)))()

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger  λ₁
x  λ₃, λ₄
id  λ₃, λ₄
comp-same  λ₅
n1
n2

Stack:
| y ↦ 1 |
| f ↦ λ₃, w ↦ 1 |
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger  λ₁
x  λ₃, λ₄
id  λ₃, λ₄
comp-same  λ₅
n1  1
n2

Stack:
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z)))()

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger              λ₁
x                    λ₃, λ₄
id                   λ₃, λ₄
comp-same            λ₅
n1                    1
n2

Stack:
  f ↦ λ₄, w ↦ 1
CFA2: pushdown automaton

(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger λ₁
x λ₃, λ₄
id λ₃, λ₄
comp-same λ₅
n1 1
n2

Stack:
| z ↦ 1 |
| f ↦ λ₄, w ↦ 1 |
(define merger
  (\lambda_1 (x) (\lambda_2 () x)))

(merger (\lambda_3 (y) y))

(define id
  (merger (\lambda_4 (z) z))())

(define comp-same
  (\lambda_5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger \lambda_1
x \lambda_3, \lambda_4
id \lambda_3, \lambda_4
comp-same \lambda_5
n1 1
n2

Stack:
\[ f \mapsto \lambda_4, w \mapsto 1 \]
CFA2: pushdown automaton

(define merger
  (\lambda_1 (x) (\lambda_2 () x)))

(merger (\lambda_3 (y) y))

(define id
  (merger (\lambda_4 (z) z))())

(define comp-same
  (\lambda_5 (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger \lambda_1
x \lambda_3, \lambda_4
id \lambda_3, \lambda_4
comp-same \lambda_5
n1 1
n2

Stack:
| z \mapsto 1
| f \mapsto \lambda_4, w \mapsto 1 |
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))))

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))
(define merger
  (λ₁ (x) (λ₂ () x)))

(merger (λ₃ (y) y))

(define id
  (merger (λ₄ (z) z))())

(define comp-same
  (λ₅ (f w) (f (f w))))

(define n1
  (comp-same id 1))

(define n2
  (comp-same id 2))

Heap:
merger λ₁
x λ₃, λ₄
id λ₃, λ₄
comp-same λ₅
n1 1
n2 2

Stack:
Resilience to syntax changes

(define id (λ (x) x))

(let* ((n1 (id 1))
        (n2 (id 2))
        (+ n1 n2))
Resilience to syntax changes

\[
\text{(define id } (\lambda (y) ((\lambda (x) x) y)))
\]

\[
\text{(let* ((n1 (id 1))}
  (n2 (id 2)))
  (+ n1 n2))
\]
Resilience to syntax changes

$$\left(\left(\lambda (id)\right)\right.
\left.\left(\text{let* } \left(\left(n1 \left(app id 1\right)\right)\right.
\left.n2 \left(app id 2\right)\right)\right.
\left.+ n1 n2\right)\)
\left(\lambda (x) x\right)\)$$
Resilience to syntax changes

(define id (λ (x) (λ () x)()))

(let* ((n1 (id 1))
        (n2 (id 2))
        (+ n1 n2))
Summarization

Functions don’t care about their return point.
Summarization

Functions don’t care about their return point.

Don’t keep track of the stack explicitly. Inside a function, remember top frame only.
Summarization

Functions don’t care about their return point.

Don’t keep track of the stack explicitly. Inside a function, remember top frame only.

Record summaries, which express in/out relations.

Use summaries at call sites to simulate the effect of the call.
CFA2: summarization

```c
main()
```

```c
42
```
CFA2: summarization

```
main()

app id 1

n1
```

```
ret := n1 + n2
```
CFA2: summarization

Callers:
2 calls  8[e \mapsto 1]
CFA2: summarization

Callers:
2 calls 8[e → 1]
CFA2: summarization

Callers:
2 calls 8[e → 1]
9[e ↦ 1] calls 12[x ↦ 1]
CFA2: summarization

Callers:
2 calls 8[e ↦→ 1]
9[e ↦→ 1] calls 12[x ↦→ 1]
CFA2: summarization

Callers:
2 calls 8[e \mapsto 1]
9[e \mapsto 1] calls 12[x \mapsto 1]

Entry/exit summaries:
12[x \mapsto 1] reaches 14[x \mapsto 1, \text{ret} \mapsto 1]
CFA2: summarization

Callers:
2 calls 8[e ↦→ 1]
9[e ↦→ 1] calls 12[x ↦→ 1]

Entry/exit summaries:
12[x ↦→ 1] reaches 14[x ↦→ 1, ret ↦→ 1]
CFA2: summarization

Callers:
2 calls 8[e \rightarrow 1]
9[e \rightarrow 1] calls 12[x \rightarrow 1]

Entry/exit summaries:
12[x \rightarrow 1] reaches 14[x \rightarrow 1, ret \rightarrow 1]
8[e \rightarrow 1] reaches 11[e \rightarrow 1, ret \rightarrow 1]
CFA2: summarization

Callers:
2  calls  8[e ↦→ 1]
9[e ↦→ 1]  calls  12[x ↦→ 1]

Entry/exit summaries:
12[x ↦→ 1]  reaches  14[x ↦→ 1, ret ↦→ 1]
8[e ↦→ 1]  reaches  11[e ↦→ 1, ret ↦→ 1]

Top level:
n1  1
CFA2: summarization

Callers:
2 calls 8[e ↦→ 1]
9[e ↦→ 1] calls 12[x ↦→ 1]

Entry/exit summaries:
12[x ↦→ 1] reaches 14[x ↦→ 1, ret ↦→ 1]
8[e ↦→ 1] reaches 11[e ↦→ 1, ret ↦→ 1]

Top level:
n1 1
CFA2: summarization

Callers:
2 calls 8[e \mapsto 1]
9[e \mapsto 1] calls 12[x \mapsto 1]
4 calls 8[e \mapsto 2]

Entry/exit summaries:
12[x \mapsto 1] reaches 14[x \mapsto 1, ret \mapsto 1]
8[e \mapsto 1] reaches 11[e \mapsto 1, ret \mapsto 1]

Top level:
n1 1
Callers:
2 calls 8[e → 1]
9[e → 1] calls 12[x → 1]
4 calls 8[e → 2]
9[e → 2] calls 12[x → 2]

Entry/exit summaries:
12[x → 1] reaches 14[x → 1, ret → 1]
8[e → 1] reaches 11[e → 1, ret → 1]
12[x → 2] reaches 14[x → 2, ret → 2]

Top level:
n1 1
CFA2: summarization

Callers:

2 calls 8[e ← 1]
9[e ← 1] calls 12[x ← 1]
4 calls 8[e ← 2]
9[e ← 2] calls 12[x ← 2]

Entry/exit summaries:

12[x ← 1] reaches 14[x ← 1, ret ← 1]
8[e ← 1] reaches 11[e ← 1, ret ← 1]
12[x ← 2] reaches 14[x ← 2, ret ← 2]
8[e ← 2] reaches 11[e ← 2, ret ← 2]

Top level:
n1 1
CFA2: summarization

Callers:
2 calls  8[e ↦→ 1]
9[e ↦ 1] calls 12[x ↦→ 1]
4 calls  8[e ↦→ 2]
9[e ↦→ 2] calls 12[x ↦→ 2]

Entry/exit summaries:
12[x ↦→ 1] reaches 14[x ↦→ 1, ret ↦→ 1]
8[e ↦→ 1] reaches 11[e ↦→ 1, ret ↦→ 1]
12[x ↦→ 2] reaches 14[x ↦→ 2, ret ↦→ 2]
8[e ↦→ 2] reaches 11[e ↦→ 2, ret ↦→ 2]

Top level:
n1 1
n2 2
CFA2: summarization

Callers:
2 calls 8[e ↔ 1]
9[e ↔ 1] calls 12[x ↔ 1]
4 calls 8[e ↔ 2]
9[e ↔ 2] calls 12[x ↔ 2]

Entry/exit summaries:
12[x ↔ 1] reaches 14[x ↔ 1, ret ↔ 1]
8[e ↔ 1] reaches 11[e ↔ 1, ret ↔ 1]
12[x ↔ 2] reaches 14[x ↔ 2, ret ↔ 2]
8[e ↔ 2] reaches 11[e ↔ 2, ret ↔ 2]

Top level:
n1 1
n2 2
ret 3
CFA2: summarization

Callers:
2 calls 8[e ↦→ 1]
9[e ↦→ 1] calls 12[x ↦→ 1]
4 calls 8[e ↦→ 2]
9[e ↦→ 2] calls 12[x ↦→ 2]

Entry/exit summaries:
12[x ↦→ 1] reaches 14[x ↦→ 1, ret ↦→ 1]
8[e ↦→ 1] reaches 11[e ↦→ 1, ret ↦→ 1]
12[x ↦→ 2] reaches 14[x ↦→ 2, ret ↦→ 2]
8[e ↦→ 2] reaches 11[e ↦→ 2, ret ↦→ 2]

Top level:
n1 1
n2 2
ret 3
Handling tail calls

(\(\text{define app } (\lambda (f \; e) (f \; e))\))
(\(\text{define id } (\lambda (x) x)\))

(let* ((n1 (\text{app id 1})))
 (n2 (\text{app id 2})))
 (+ n1 n2))
Handling tail calls

\[
\begin{align*}
&\text{(define app } (\lambda (f \ e) (f \ e))) \\
&(\text{define id } (\lambda (x) x)) \\
&(\text{let* } ((n1 (app id 1))) \\
&\quad (n2 (app id 2))) \\
&\quad (+ \ n1 \ n2))
\end{align*}
\]
(define app (λ (f e) (f e)))
(define id (λ (x) x))

(let* ((n1 (app id 1))
        (n2 (app id 2)))
  (+ n1 n2))

With tail calls, call site and return point in different procedures.
Handling tail calls

(define app (\(f\ e\) \(f\ e\)))
(define id (\(x\) x))

(let* ((n1 (app id 1))
        (n2 (app id 2)))
  (+ n1 n2))

With tail calls, call site and return point in different procedures.

Cross-procedure summaries:
From entry of app to exit of id.
Handling first-class control

Summarization relies on call/return nesting. As a result, it can’t handle generators, coroutines, call/cc.
Handling first-class control

Summarization relies on call/return nesting. As a result, it can’t handle generators, coroutines, call/cc.

Restricted CPS:

\[(\lambda_1 (f \text{ cc}) (f (\lambda_2 (u \text{ k}) (\text{cc u})) \text{ cc})) \quad \checkmark\]

\[(\lambda_1 (f \text{ cc}) (f (\lambda_2 (u \text{ k}) (u 123 \text{ cc})) \text{ cc})) \quad \times\]
Handling first-class control

Summarization relies on call/return nesting. As a result, it can’t handle generators, coroutines, call/cc.

Restricted CPS:

\[
(\lambda_1 \ (f \ cc) \ (f \ (\lambda_2 \ (u \ k) \ (cc \ u)) \ cc)) \quad \checkmark
\]
\[
(\lambda_1 \ (f \ cc) \ (f \ (\lambda_2 \ (u \ k) \ (u \ 123 \ cc)) \ cc)) \quad \times
\]

Effective stack reasoning in the presence of first-class control. Summaries for call/cc: connect entry of \(\lambda_1\) with call (cc u).

All kinds of summaries (normal call/return, tail calls, exceptions, first-class control) connect a continuation passed to a user function with the state that calls it.
Theoretical formulation of CFA2

Abstract interpretation of CPS programs (1st-class control).

Concrete semantics [Might 07]
- Expose stack structure

Abstract semantics
- Orbit stack policy
- Stack and heap environments
- Stack and heap references
- Nothing tricky here

Local semantics
- No stack.
- Generalized summaries (tail calls, call/cc).
- Record callers as you find them.
Correctness

Simulation
The abstract semantics is a safe approximation of the runtime behavior of the program.
Correctness

Simulation
The abstract semantics is a safe approximation of the runtime behavior of the program.

Soundness
The summarization algorithm doesn’t miss any flows of the abstract semantics . . .
Correctness

Simulation
The abstract semantics is a safe approximation of the runtime behavior of the program.

Soundness
The summarization algorithm doesn’t miss any flows of the abstract semantics . . .

Completeness
. . . and it doesn’t add spurious flows.
The only composite piece of data is the object. Functions, arrays are objects.

Object: map from strings (property names) to values. Properties can be added/deleted at runtime. Full field sensitivity undecidable.

Inheritance: each object has one prototype object. No cycles in the prototype chain.
Array access: a[i]
General computed-property access: obj[prop]
Static analysis for JavaScript

Array access: \(a[i]\)
General computed-property access: \(obj[prop]\)

Objects can have many prototypes.
The prototype chain can have cycles.
Static analysis for JavaScript

Array access: \( a[i] \)
General computed-property access: \( \text{obj}[\text{prop}] \)

Objects can have many prototypes.
The prototype chain can have cycles.

Exceptions are included in the summaries.
Static analysis for JavaScript

Array access: a[i]
General computed-property access: obj[prop]

Objects can have many prototypes.
The prototype chain can have cycles.

Exceptions are included in the summaries.

Recursive implementation of call/return matching (ask me).
Analyzing Firefox add-ons

Core JavaScript manageable in a summer. Inferred types for Sunspider, V8 benchmarks. DOM is huge.
Analyzing Firefox add-ons

Core JavaScript manageable in a summer.
Inferred types for Sunspider, V8 benchmarks.
DOM is huge.

Chrome: Firefox elements, add-on elements
Content: Webpage elements
Analyzing Firefox add-ons

Core JavaScript manageable in a summer.
Inferred types for Sunspider, V8 benchmarks.
DOM is huge.

Chrome: Firefox elements, add-on elements
Content: Webpage elements

Events can be generated from chrome/content.
Listeners can be attached to chrome/content.
New architecture prevents listening on chrome for content.
## Results

<table>
<thead>
<tr>
<th>Extension</th>
<th>LOC</th>
<th>time (ms)</th>
<th>safe/total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commentblocker</td>
<td>537</td>
<td>248</td>
<td>3/10</td>
</tr>
<tr>
<td>Flashblock</td>
<td>935</td>
<td>357</td>
<td>3/5</td>
</tr>
<tr>
<td>Imtranslator</td>
<td>1263</td>
<td>406</td>
<td>2/4</td>
</tr>
<tr>
<td>Flagfox</td>
<td>2081</td>
<td>896</td>
<td>5/12</td>
</tr>
<tr>
<td>Greasemonkey</td>
<td>4809</td>
<td>1716</td>
<td>13/23</td>
</tr>
<tr>
<td>Flashgot</td>
<td>9741</td>
<td>4524</td>
<td>10/21</td>
</tr>
<tr>
<td>Video download helper</td>
<td>12749</td>
<td>4621</td>
<td>13/19</td>
</tr>
<tr>
<td>Web developer</td>
<td>22018</td>
<td>12603</td>
<td>9/63</td>
</tr>
<tr>
<td>Stumbleupon</td>
<td>32594</td>
<td>18235</td>
<td>13/44</td>
</tr>
</tbody>
</table>
ToDo list

CFA2:
- Complexity? Polytime variant?
- Completeness for first-class control?
ToDo list

CFA2:
- Complexity? Polytime variant?
- Completeness for first-class control?

Finite-state vs type-based vs pushdown
ToDo list

CFA2:
- Complexity? Polytime variant?
- Completeness for first-class control?

Finite-state vs type-based vs pushdown

Declarative specification of an analysis (Jones–Muchnick vision)
Todo list

CFA2:
- Complexity? Polytime variant?
- Completeness for first-class control?

Finite-state vs type-based vs pushdown

Declarative specification of an analysis (Jones–Muchnick vision)

Polyvariant CFA should be very efficient
- if not much recursion/loops in $p$, then a bit slower than $p$.
- if recursion/loops in $p$, then much faster than $p$. 
More info

Slides: www.ccs.neu.edu/home/dimvar/cfa2-shonan.pdf

CFA2 w/out first-class control [ESOP 10, LMCS 11]

Restricted CPS [PEPM 11]

CFA2 w/ first-class control [ICFP 11]

DoctorJS: github.com/mozilla/doctorjs