CFA2: Pushdown Flow Analysis for Higher-Order Languages

Dimitris Vardoulakis

Northeastern University

Flow analysis is instrumental in building good software.

Optimization

Debugging

Verification

Development

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.
$\Rightarrow$ executions are strings in a regular language.
$\Rightarrow$ approximate program with finite-state machine.
$\Rightarrow$ call/return mismatch.

Finite-state analyses

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

Fine for conditionals and loops (think Fortran).

Finite-state analyses

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

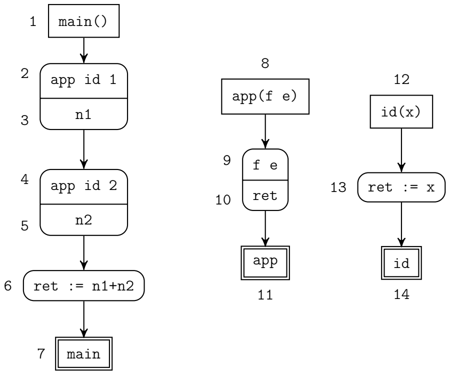
Fine for conditionals and loops (think Fortran).

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

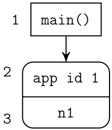


0CFA example

```
main()
```

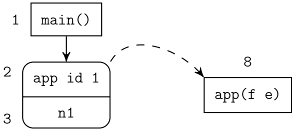
Global environment:

0CFA example



Global environment:

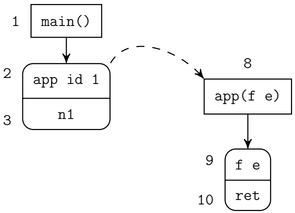
0CFA example



Global environment:

f	id
e	1

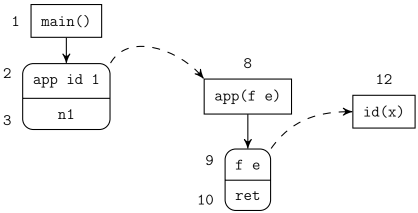
0CFA example



Global environment:

f	id
e	1

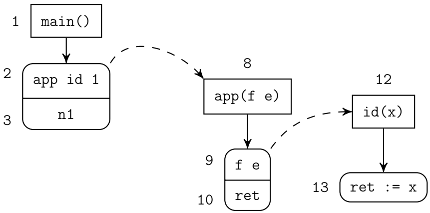
0CFA example



Global environment:

f	id
e	1
x	1

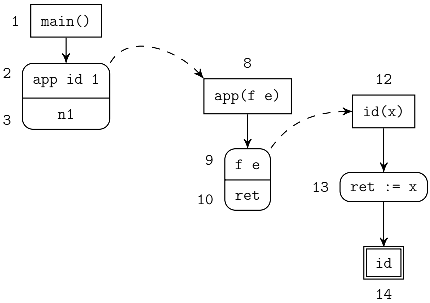
0CFA example



Global environment:

f	id
e	1
x	1
ret-id	1

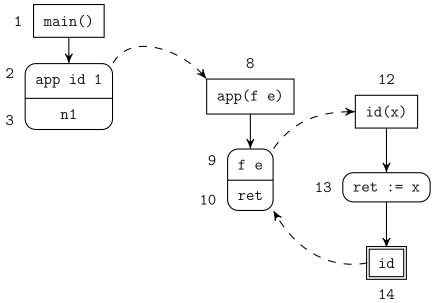
0CFA example



Global environment:

f	id
e	1
x	1
ret-id	1

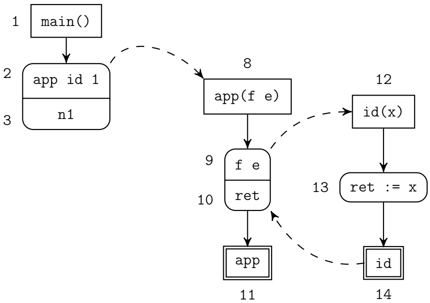
0CFA example



Global environment:

f	id
e	1
ret-app	1
x	1
ret-id	1

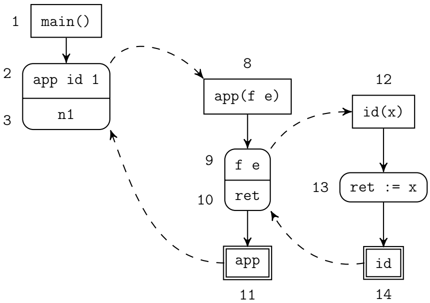
0CFA example



Global environment:

f	id
e	1
ret-app	1
x	1
ret-id	1

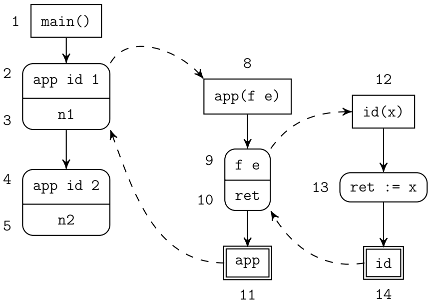
0CFA example



Global environment:

n1	1
f	id
e	1
ret-app	1
x	1
ret-id	1

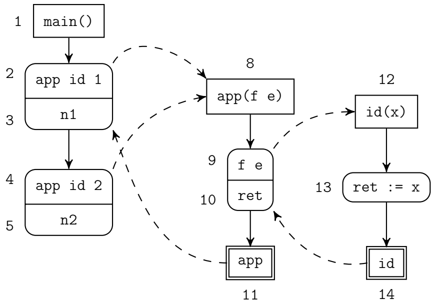
0CFA example



Global environment:

n1	1
f	id
e	1
ret-app	1
x	1
ret-id	1

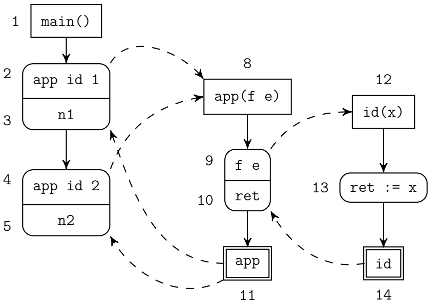
0CFA example



Global environment:

n1	1	
f	id	
e	1	2
ret-app	1	2
x	1	2
ret-id	1	2

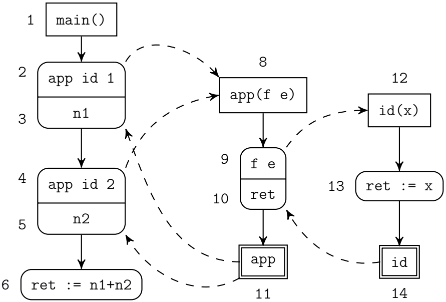
0CFA example



Global environment:

n1	1	2
n2	1	2
f	id	
e	1	2
ret-app	1	2
x	1	2
ret-id	1	2

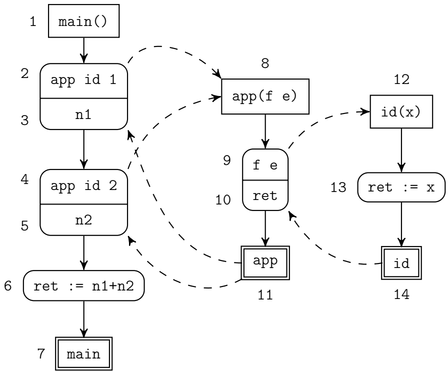
0CFA example



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	

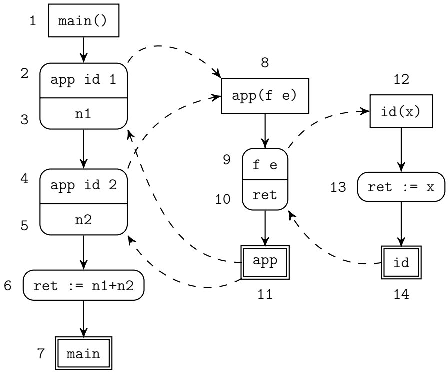
0CFA example



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	

0CFA example

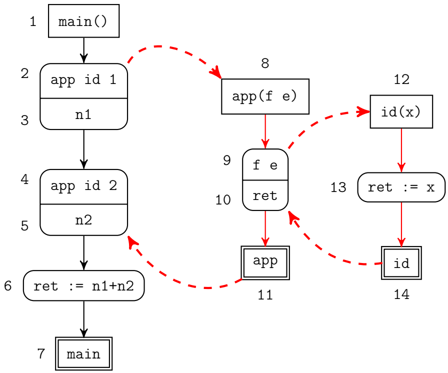


Global environment:

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.

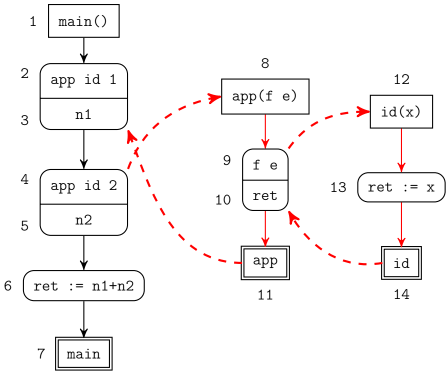
0CFA example



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	

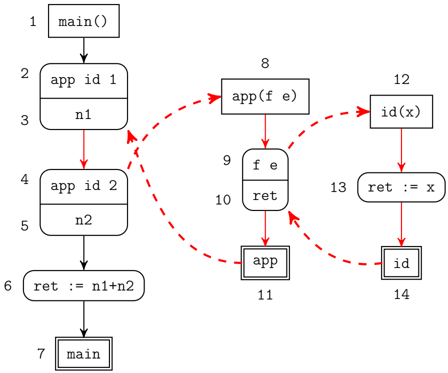
0CFA example



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	

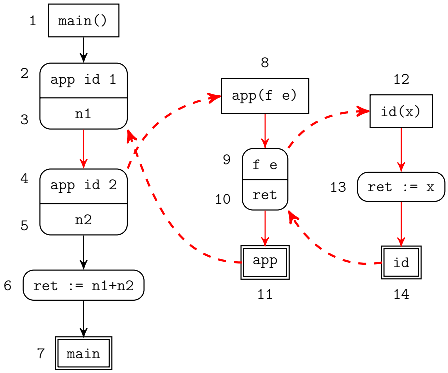
0CFA example



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
$\Rightarrow$ cannot accurately calculate stack change.

Fake rebinding

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

Fake rebinding

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

λ_a λ_b

Fake rebinding

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

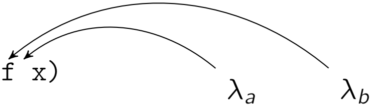
λ_a λ_b

Flows:

```
(f (f x))
```

Fake rebinding

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



Flows:

(f (λ_a x))

Fake rebinding

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

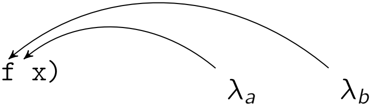
λ_a λ_b

Flows:

$(\lambda_a\ (\lambda_a\ \texttt{x}))$ ✓

Fake rebinding

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



Flows:

(λ_a (λ_a x)) ✓

(λ_b (λ_b x)) ✓

Fake rebinding

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

λ_a λ_b

Flows:

(λ_a (λ_a x)) ✓

(λ_b (λ_b x)) ✓

(λ_b (λ_a x)) ✗

Fake rebinding

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

λ_a λ_b

Flows:

(λ$_a$ (λ$_a$ x)) ✓

(λ$_b$ (λ$_b$ x)) ✓

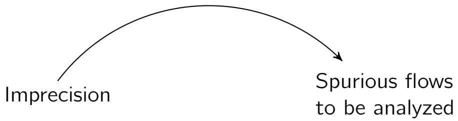
(λ$_b$ (λ$_a$ x)) ✗

(λ$_a$ (λ$_b$ x)) ✗

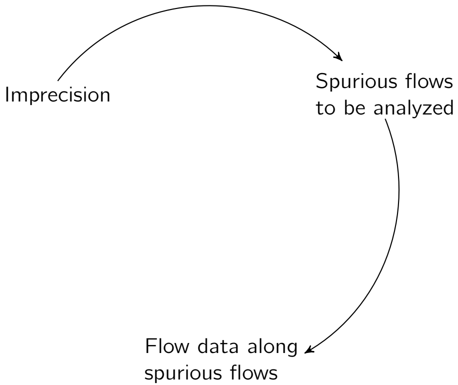
Imprecision slows down the analysis

Imprecision

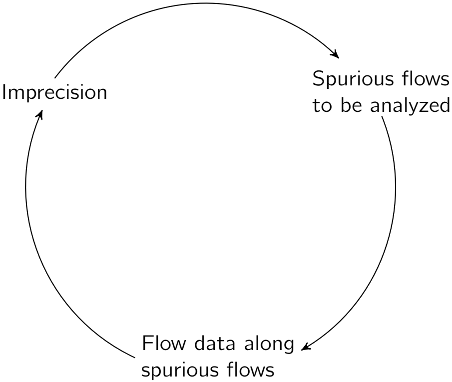
Imprecision slows down the analysis



Imprecision slows down the analysis



Imprecision slows down the analysis



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
$\Rightarrow$ infinite state space.

What we hope to achieve

Advanced reasoning about stack and environment:

- escape analysis for stack allocation
- super-β inlining
- transducer fusion

What we hope to achieve

Advanced reasoning about stack and environment:

- escape analysis for stack allocation
- super-β 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: `(λ(x) (λ(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: `(λ(x) (λ(y) (y (y x))))`
Bound in the top frame.
Stack references of same variable bound in same environment.

Heap references: `(λ(x) (λ(y) (y (y x))))`
Either deeper in stack or in heap.

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
  (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	λ_1
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	λ_1
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	λ_1
x	λ_3
id	
comp-same	
n1	
n2	

Stack:

x $\mapsto \lambda_3$

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	λ_1
x	λ_3
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	λ_1
x	λ_3, λ_4
id	
comp-same	
n1	
n2	

Stack:

x $\mapsto$ λ_4

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	λ_1
x	λ_3, λ_4
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	λ_1
x	λ_3, λ_4
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	
n2	

Stack:

$\texttt{f} \mapsto \{\lambda_3, \lambda_4\}, \texttt{w} \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	
n2	

Stack:

$f \mapsto \lambda_3,\ w \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	
n2	

Stack:

y $\mapsto$ 1
f $\mapsto$ λ_3, w $\mapsto$ 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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	
n2	

Stack:

f $\mapsto \lambda_3$, w $\mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	
n2	

Stack:

$y \mapsto 1$
$f \mapsto \lambda_3$, $w \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
n2	

Stack:

$f \mapsto \lambda_4,\ w \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
n2	

Stack:

$z \mapsto 1$
$f \mapsto \lambda_4$, $w \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
n2	

Stack:

f $\mapsto \lambda_4$, w $\mapsto$ 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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
n2	

Stack:

$z \mapsto 1$
$f \mapsto \lambda_4$, $w \mapsto 1$

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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
n1	1
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	λ_1
x	λ_3, λ_4
id	λ_3, λ_4
comp-same	λ_5
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

```
(define id (λ (y) ((λ (x) x) y)))

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

Resilience to syntax changes

```
((λ (id)
   (let* ((n1 (app id 1))
          (n2 (app id 2)))
     (+ n1 n2)))
 (λ (x) x))
```

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.

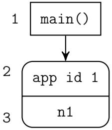
Record summaries, which express in/out relations.

Use summaries at call sites to simulate the effect of the call.

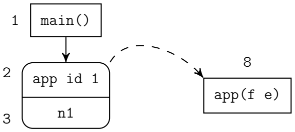
CFA2: summarization

1 `main()`

CFA2: summarization



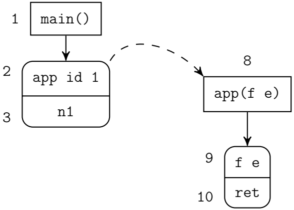
CFA2: summarization



Callers:

2 calls $8[e \mapsto 1]$

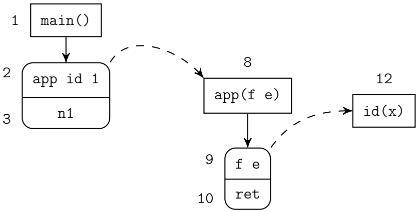
CFA2: summarization



Callers:

2 calls $8[e \mapsto 1]$

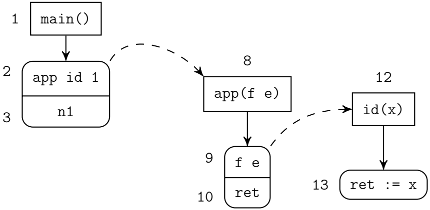
CFA2: summarization



Callers:

2	calls	8[e ↦ 1]
9[e ↦ 1]	calls	12[x ↦ 1]

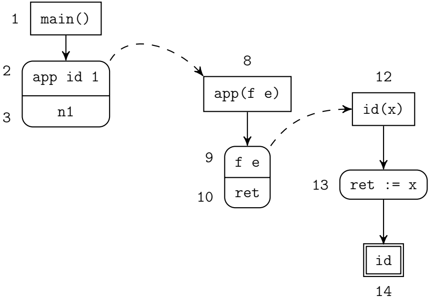
CFA2: summarization



Callers:

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

CFA2: summarization



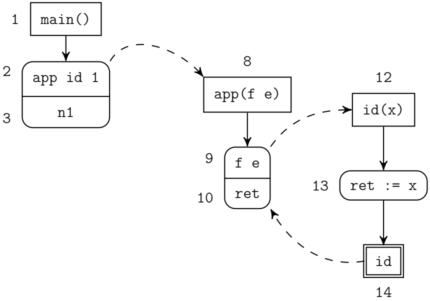
Callers:

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, ret $\mapsto$ 1]

CFA2: summarization



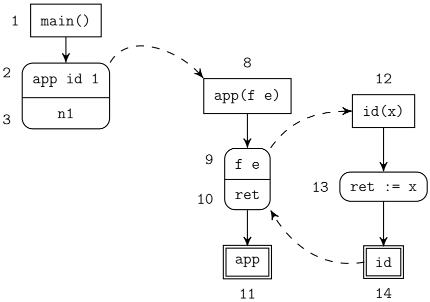
Callers:

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



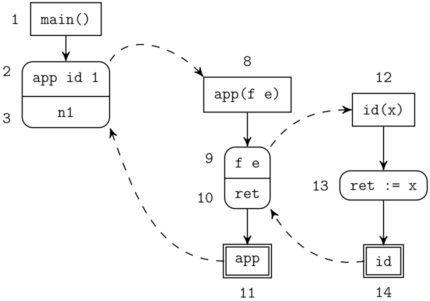
Callers:

2	calls	8[e ↦ 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]

CFA2: summarization



Callers:

2	calls	8[e ↦ 1]
9[e ↦ 1]	calls	12[x ↦ 1]

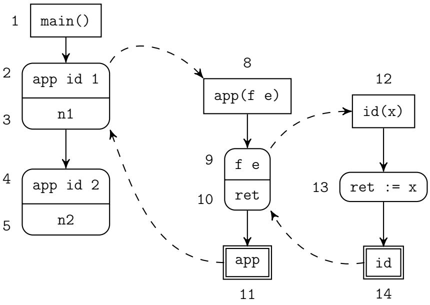
Entry/exit summaries:

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]

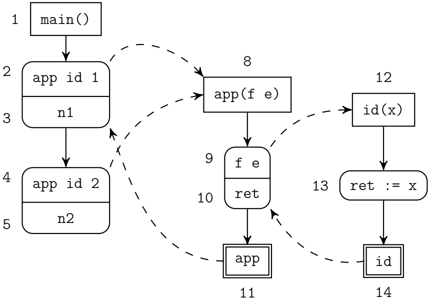
Entry/exit summaries:

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

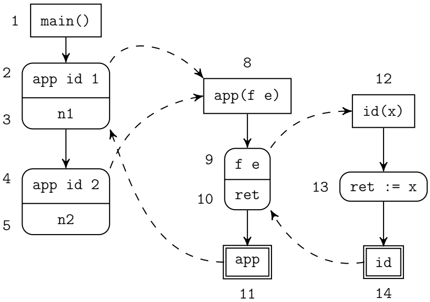
Entry/exit summaries:

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]
4	calls	8[e ↦ 2]
9[e ↦ 2]	calls	12[x ↦ 2]

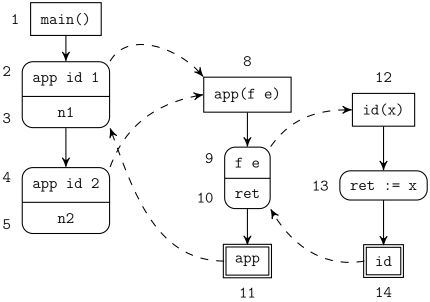
Entry/exit summaries:

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

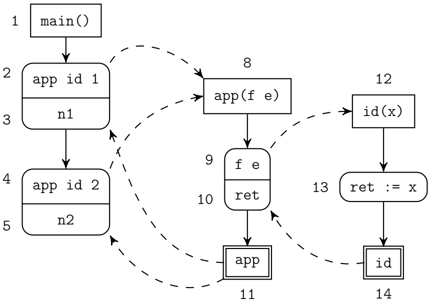
Entry/exit summaries:

$12[x \mapsto 1]$	reaches	$14[x \mapsto 1, \texttt{ret} \mapsto 1]$
$8[e \mapsto 1]$	reaches	$11[e \mapsto 1, \texttt{ret} \mapsto 1]$
$12[x \mapsto 2]$	reaches	$14[x \mapsto 2, \texttt{ret} \mapsto 2]$
$8[e \mapsto 2]$	reaches	$11[e \mapsto 2, \texttt{ret} \mapsto 2]$

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]

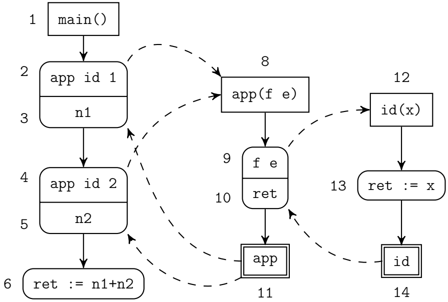
Entry/exit summaries:

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]

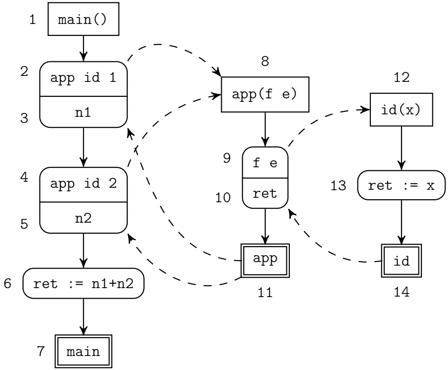
Entry/exit summaries:

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 $\mapsto$ 1]
9[e $\mapsto$ 1]	calls	12[x $\mapsto$ 1]
4	calls	8[e $\mapsto$ 2]
9[e $\mapsto$ 2]	calls	12[x $\mapsto$ 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]
12[x $\mapsto$ 2]	reaches	14[x $\mapsto$ 2, ret $\mapsto$ 2]
8[e $\mapsto$ 2]	reaches	11[e $\mapsto$ 2, ret $\mapsto$ 2]

Top level:

n1	1
n2	2
ret	3

Handling tail calls

```
(define app (λ (f e) (f e)))
(define id (λ (x) x))

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

Handling tail calls

```
(define app (λ (f e) (f e)))
(define id (λ (x) x))

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

Handling tail calls

```
(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 (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.

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:

```
(λ1 (f cc) (f (λ2 (u k) (cc u)) cc))          ✓
(λ1 (f cc) (f (λ2 (u k) (u 123 cc)) 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 cc) (f (`λ_2` (u k) (cc u)) cc))` ✓

$(\lambda_1$ `(f cc) (f (`λ_2` (u k) (u 123 cc)) cc))` ✗

Effective stack reasoning in the presence of first-class control.
Summaries for `call/cc`: connect entry of λ_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]

$\Downarrow$ expose stack structure

Abstract semantics
- Orbit stack policy
- Stack and heap environments
- Stack and heap references

$\Downarrow$ nothing tricky here

Local semantics + summarization

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.

JavaScript

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.

Static analysis for JavaScript

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: `obj[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

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

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`