CFA2: a Context-Free Approach to Control-Flow Analysis

Dimitrios Vardoulakis   Olin Shivers

Northeastern University

What is a flow analysis?

Flow analysis: find information about the control and data flow of a program without running it.

Applications

Bug finding

argument mismatch
type mismatch
array-index out of bounds
dead-code detection

Semantic navigation

what functions get called at this call site
what flows to this variable

Optimization

classic dataflow optimizations
function-call resolution
type recovery for tag elimination

From graphs to pushdown models

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.

From graphs to pushdown models

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.

Fine for conditionals and loops (think Fortran).
Weak for first-class functions.

From graphs to pushdown models

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.

Fine for conditionals and loops (think Fortran).
Weak for first-class functions.

Approximate program with pushdown automaton.
$\Rightarrow$ unbounded call/return matching.

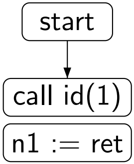
```
(define id (λ(x) x))

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

0CFA execution

start

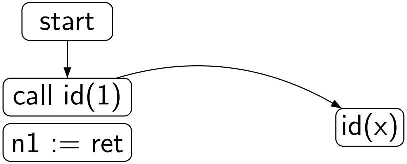
Global environment:

0CFA execution



Global environment:

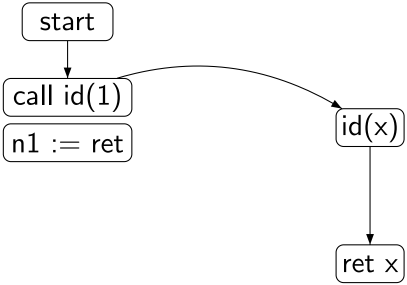
0CFA execution



Global environment:

x	1

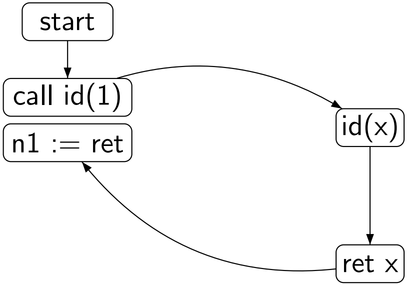
0CFA execution



Global environment:

x	1

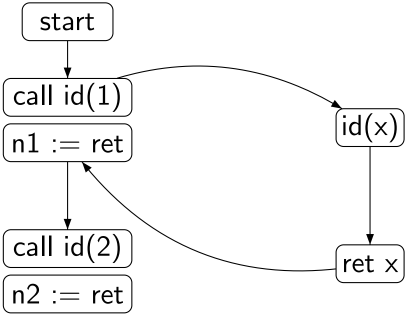
0CFA execution



Global environment:

x	1
n1	1

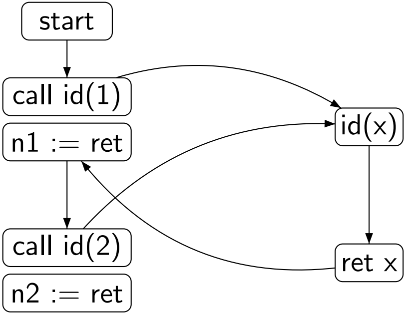
0CFA execution



Global environment:

x	1
n1	1

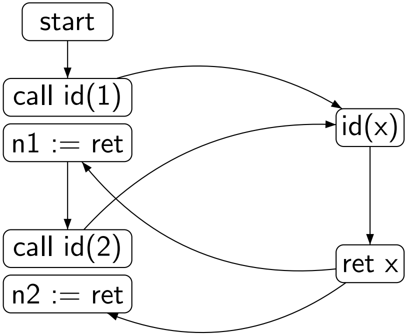
0CFA execution



Global environment:

x	1	2
n1	1	

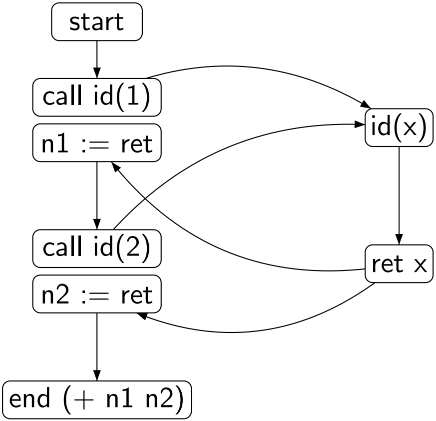
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

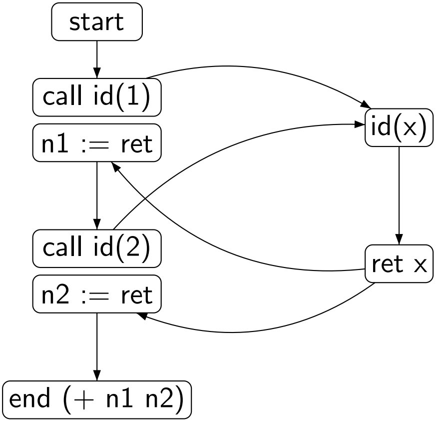
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

0CFA execution

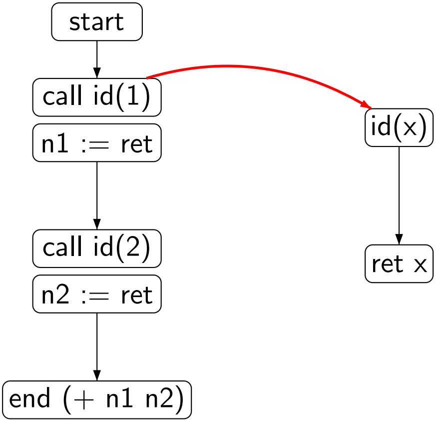


Global environment:

x	1	2
n1	1	2
n2	1	2

Call/return mismatch causes spurious flows
⇒ commonly called functions pollute the analysis.

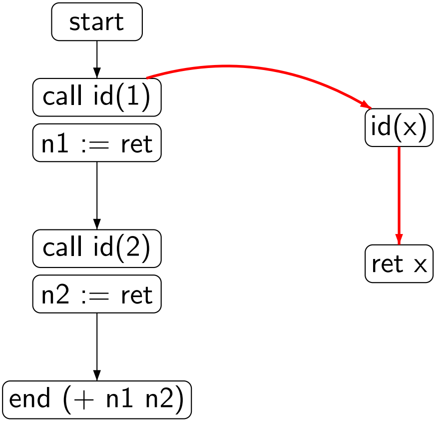
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

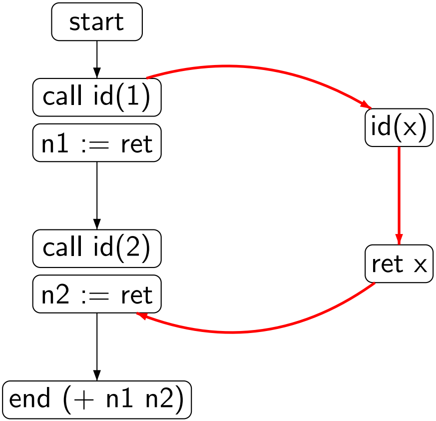
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

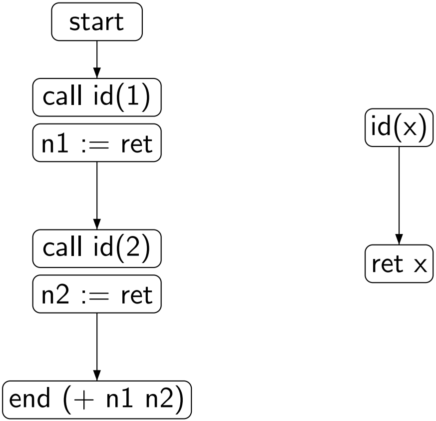
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

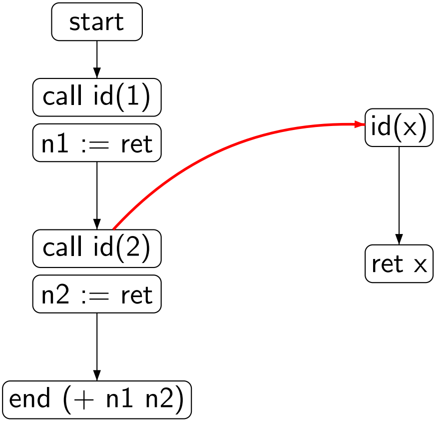
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

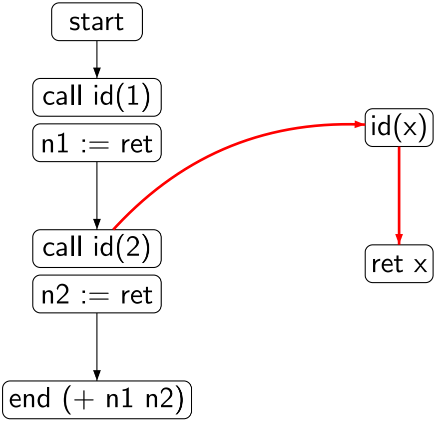
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

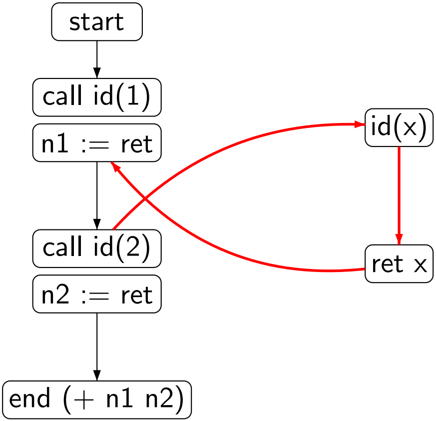
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

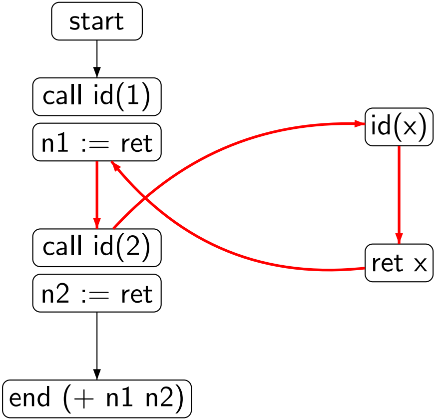
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

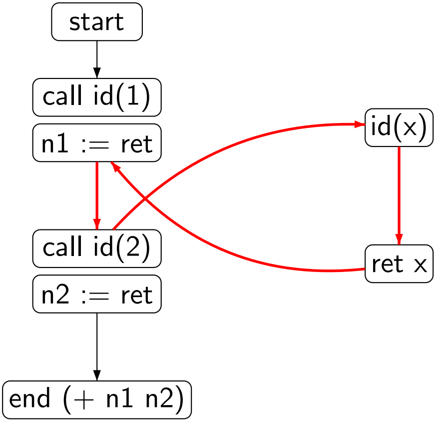
0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

0CFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

Can't use a graph model to calculate stack change
⇒ stack-based optimizations out of reach.

Fake Rebinding

```
(define (compose-same f x)
  ^2(f ^1(f x)))
```

Fake Rebinding

(define (compose-same f x) λ_a λ_b

^{2}(f ^{1}(f x)))

Fake Rebinding

`(define (compose-same f x)` λ_a λ_b

^{2}`(f` ^{1}`(f x)))`

Flows:

^{2}`(f` ^{1}`(f x))`

Fake Rebinding

```
(define (compose-same f x)                 λa          λb
  2(f 1(f x)))
```

Flows:

^{2}(f ^{1}(λ_a x))

Fake Rebinding

`(define (compose-same f x)` λ_a λ_b

^{2}`(f` ^{1}`(f x)))`

Flows:

^{2}`(`λ_a ^{1}`(`λ_a `x))` ✓

Fake Rebinding

`(define (compose-same f x)` λ_a λ_b

^{2}`(f` ^{1}`(f x)))`

Flows:

$^{2}(\lambda_a\ ^{1}(\lambda_a$ `x))` ✓

$^{2}(\lambda_b\ ^{1}(\lambda_b$ `x))` ✓

Fake Rebinding

(define (compose-same f x) λ_a λ_b

^{2}(f ^{1}(f x)))

Flows:

$^{2}(\lambda_a \; ^{1}(\lambda_a$ x)) ✓

$^{2}(\lambda_b \; ^{1}(\lambda_b$ x)) ✓

$^{2}(\lambda_b \; ^{1}(\lambda_a$ x)) ✗

Fake Rebinding

```
(define (compose-same f x)
  ²(f ¹(f x)))
```

λ_a λ_b

Flows:

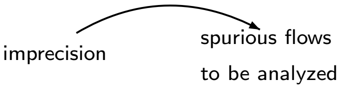
$^{2}(\lambda_a\ ^{1}(\lambda_a\ \mathtt{x}))$ ✓

$^{2}(\lambda_b\ ^{1}(\lambda_b\ \mathtt{x}))$ ✓

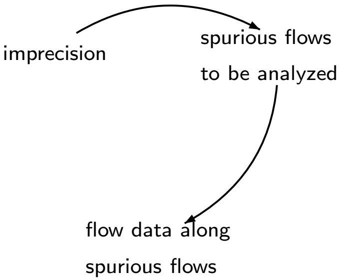
$^{2}(\lambda_b\ ^{1}(\lambda_a\ \mathtt{x}))$ ✗

$^{2}(\lambda_a\ ^{1}(\lambda_b\ \mathtt{x}))$ ✗

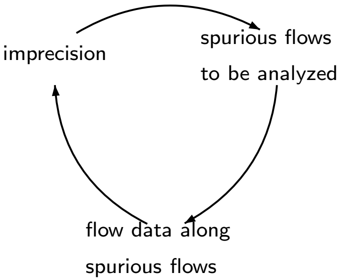
The vicious cycle of approximation



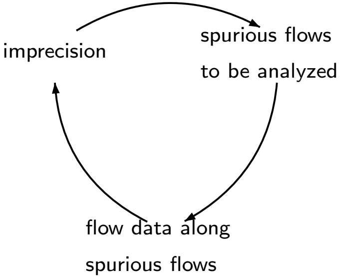
The vicious cycle of approximation



The vicious cycle of approximation



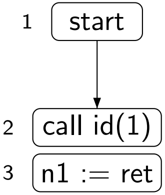
The vicious cycle of approximation



- In HOFA, imprecision can increase running time.
- *k*-CFA intractably slow for $k > 0$ (Van Horn–Mairson).

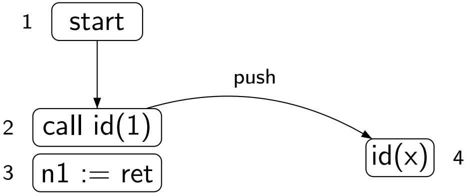
CFA2: pushdown automaton

CFA2: pushdown automaton



Stack:

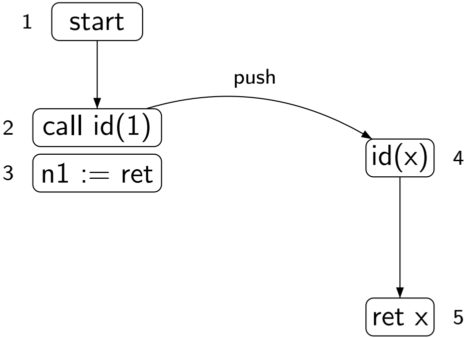
CFA2: pushdown automaton



Stack:

| x ↦ 1, ret ↦ 3 |

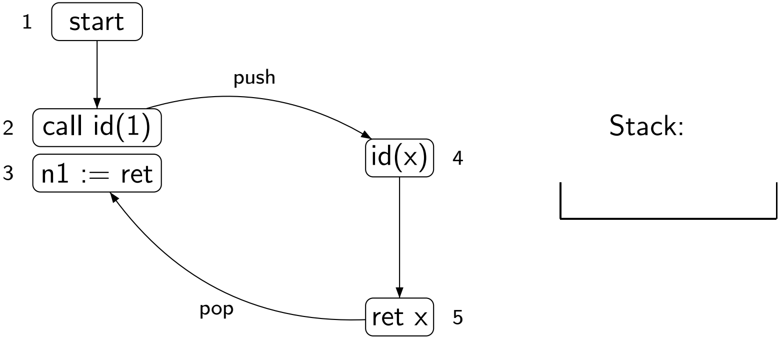
CFA2: pushdown automaton



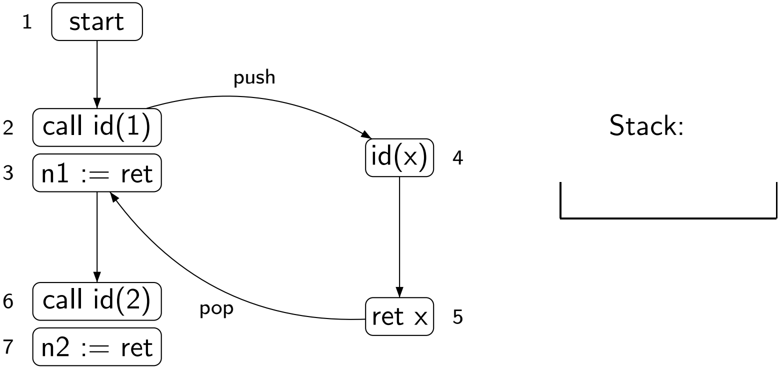
Stack:

x ↦ 1, ret ↦ 3

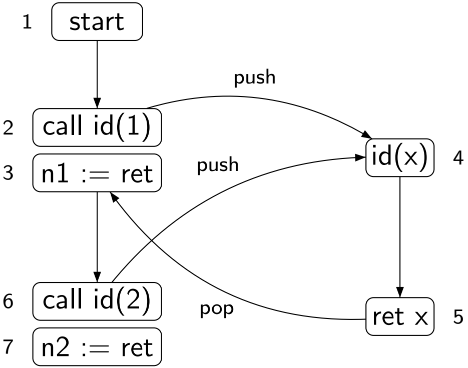
CFA2: pushdown automaton



CFA2: pushdown automaton



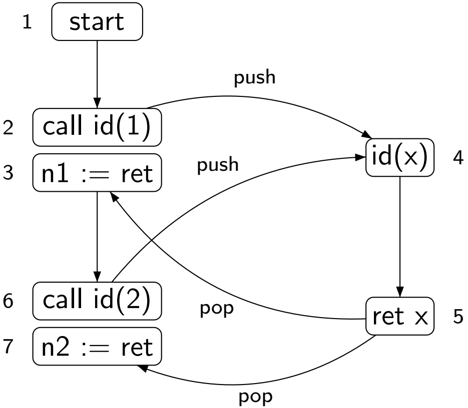
CFA2: pushdown automaton



Stack:

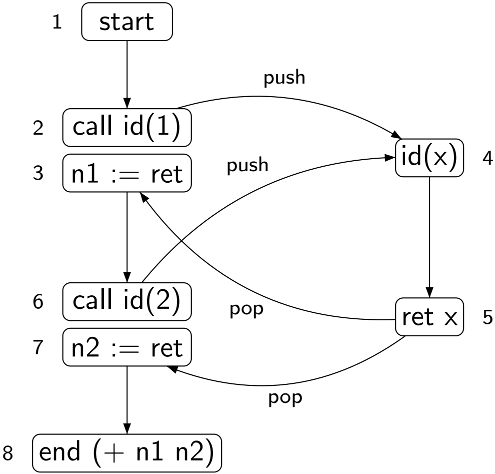
$x \mapsto 2$, $ret \mapsto 7$

CFA2: pushdown automaton



Stack:

CFA2: pushdown automaton



Stack:

Summarization

Why:

Stack can grow arbitrarily—infinite state space
$\Rightarrow$ simple analysis techniques won't terminate!

Summarization handles infinite-space issue (Sharir–Pnueli, Reps).

Summarization

Why:

Stack can grow arbitrarily—infinite state space
$\Rightarrow$ simple analysis techniques won't terminate!

Summarization handles infinite-space issue (Sharir–Pnueli, Reps).

How:

On function entry, forget return point; remember before return.

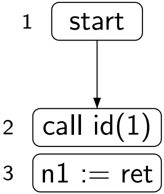
Inside a function, remember only the top frame.

To avoid reanalyzing functions often, record summaries from function entries to function exits.

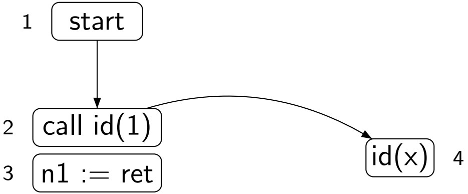
CFA2: summarization

1 start

CFA2: summarization

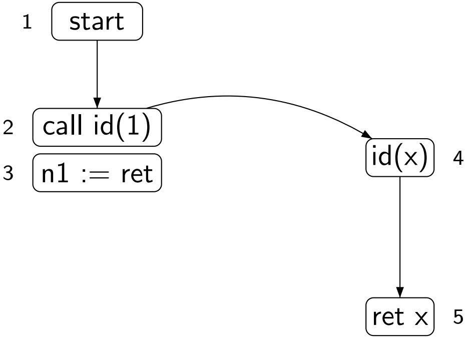


CFA2: summarization



Callers:
2 calls 4[x $\mapsto$ 1]

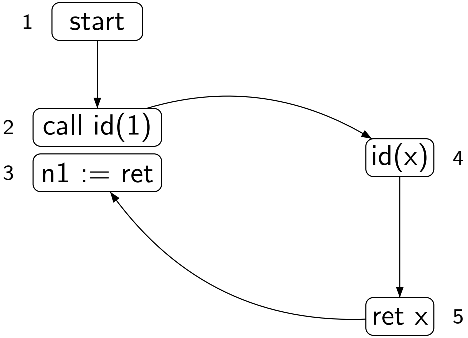
CFA2: summarization



Callers:
2 calls 4[x $\mapsto$ 1]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]

CFA2: summarization

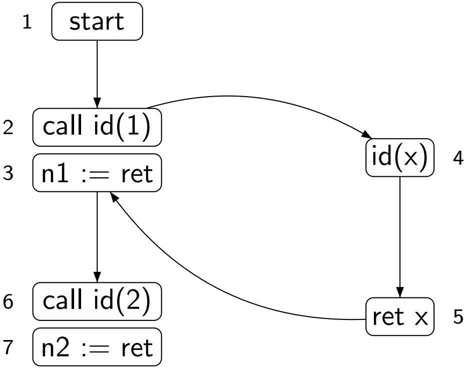


Callers:
2 calls 4[x $\mapsto$ 1]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]

Toplevel:
n1 1

CFA2: summarization

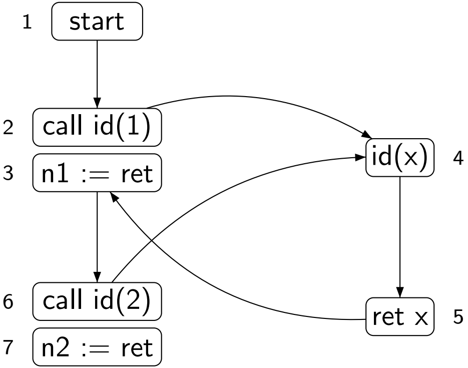


Callers:
2 calls 4[x $\mapsto$ 1]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]

Toplevel:
n1 1

CFA2: summarization

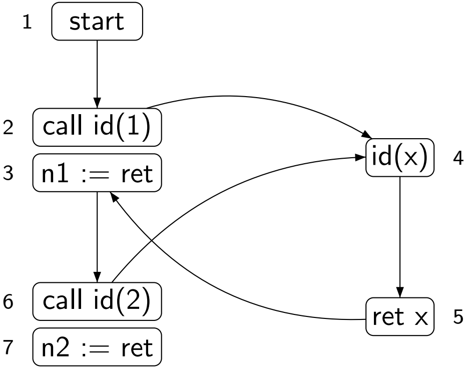


Callers:
2 calls 4[x $\mapsto$ 1]
6 calls 4[x $\mapsto$ 2]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]

Toplevel:
n1 1

CFA2: summarization

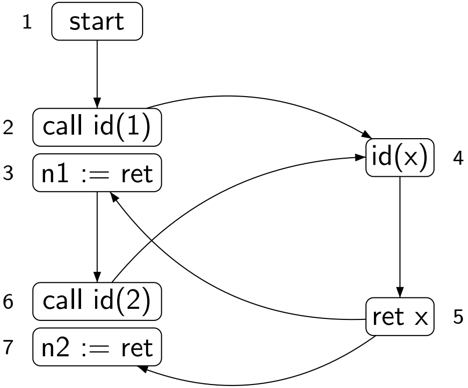


Callers:
2 calls 4[x $\mapsto$ 1]
6 calls 4[x $\mapsto$ 2]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]
4[x $\mapsto$ 2] goes to 5[x $\mapsto$ 2]

Toplevel:
n1 1

CFA2: summarization



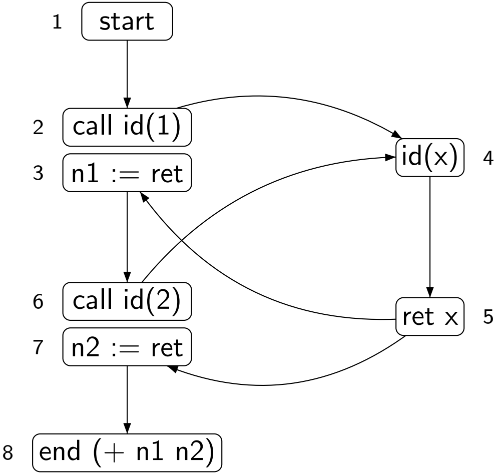
Callers:
2 calls 4[x $\mapsto$ 1]
6 calls 4[x $\mapsto$ 2]

Summaries:
4[x $\mapsto$ 1] goes to 5[x $\mapsto$ 1]
4[x $\mapsto$ 2] goes to 5[x $\mapsto$ 2]

Toplevel:

n1	1
n2	2

CFA2: summarization



Callers:
2 calls 4[x ↦ 1]
6 calls 4[x ↦ 2]

Summaries:
4[x ↦ 1] goes to 5[x ↦ 1]
4[x ↦ 2] goes to 5[x ↦ 2]

Toplevel:

n1	1
n2	2

Stack filtering

(define (compose-same f x) λ_a λ_b

^{2}(f ^{1}(f x)))

Stack filtering

```
(define (compose-same f x)
  ^2(f ^1(f x)))
```

λ_a λ_b

Flows:	Frame:	Action:
`^2(f ^1(f x))`	[f $\mapsto \{\lambda_a, \lambda_b\}$]	pick a lambda

Stack filtering

```
(define (compose-same f x)
  ²(f ¹(f x)))
```

λ_a λ_b

Flows:	Frame:	Action:
`²(f ¹(f x))`	$[\mathrm{f} \mapsto \{\lambda_a, \lambda_b\}]$	pick a lambda
`²(f ¹(`λ_a` x))`	$[\mathrm{f} \mapsto \{\lambda_a\}]$	commit to λ_a

Stack filtering

(define (compose-same f x)
^{2}(f ^{1}(f x)))

λ_a λ_b

Flows:	Frame:	Action:
^{2}(f ^{1}(f x))	$[\mathsf{f} \mapsto \{\lambda_a, \lambda_b\}]$	pick a lambda
^{2}(f ^{1}(λ_a x))	$[\mathsf{f} \mapsto \{\lambda_a\}]$	commit to λ_a
^{2}(λ_a ^{1}(λ_a x))	$[\mathsf{f} \mapsto \{\lambda_a\}]$	

Stack filtering

`(define (compose-same f x)` λ_a λ_b

^{2}`(f` ^{1}`(f x)))`

Flows:	Frame:	Action:
^{2}(f ^{1}(f x))	$[\mathsf{f} \mapsto \{\lambda_a, \lambda_b\}]$	pick a lambda
^{2}(f ^{1}(λ_a x))	$[\mathsf{f} \mapsto \{\lambda_a\}]$	commit to λ_a
^{2}(λ_a ^{1}(λ_a x))	$[\mathsf{f} \mapsto \{\lambda_a\}]$	

Similarly for λ_b

Stack and heap references

Stack filtering possible because both references to f in top frame (stack references).

Stack and heap references

Stack filtering possible because both references to f in top frame (stack references).

Some references not in top frame though (heap references), e.g., `(λ(x)(λ(y) (+ x y)))`

Characteristics of CFA2

- handles first-class functions, tail calls.
- unbounded call/return matching.
- applies to statically typed and dynamic languages.
- precise lookup for stack references.
- strong update for stack references.

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.

Benchmarks

	$S_?$	$H_?$	0CFA		1CFA		CFA2	
			visited	const	visited	const	visited	const
`len`	9	0	81	0	126	0	55	2
`rev-iter`	17	0	121	0	198	0	82	4
`len-Y`	15	4	199	0	356	0	131	2
`tree-count`	33	0	293	2	2856	6	183	10
`ins-sort`	33	5	509	0	1597	0	600	4
`DFS`	94	11	1337	8	6890	8	1709	16
`flatten`	37	0	1520	0	6865	0	478	5
`sets`	90	3	3915	0	54414	0	4233	4
`church-nums`	46	23	19130	0	19411	0	24580	0

Future work

- `call/cc`
- increase precision in heap (Might–Shivers ΓCFA).
- escape analysis: stack allocation of closures, cons cells etc.

Conclusions

- Flow-analysis of higher-order languages has captivated researchers for the past 20 years.
- CFA2 models well the important control-flow structure of these languages: function call/return.
- Exciting possibilities opening up:
  - optimization
  - informative development environments
  - compile-time error detection

Thank you!