

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.

⇒ executions are strings in a regular language.

⇒ approximate program with finite-state machine.

From graphs to pushdown models

Program as a graph whose nodes are the program points.

⇒ executions are strings in a regular language.

⇒ 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.

⇒ executions are strings in a regular language.

⇒ approximate program with finite-state machine.

Fine for conditionals and loops (think Fortran).

Weak for first-class functions.

Approximate program with pushdown automaton.

⇒ unbounded call/return matching.

```
(define id ( $\lambda(x)$  x))
```

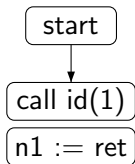
```
(let* ((n1 (id 1))  
      (n2 (id 2)))  
  (+ n1 n2))
```

OCFA execution

start

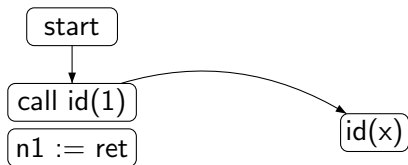
Global environment:

OCFA execution



Global environment:

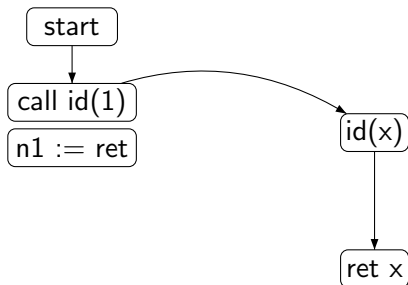
OCFA execution



Global environment:

x 1

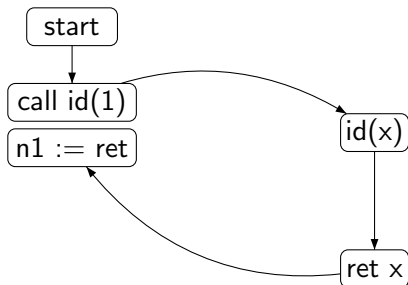
OCFA execution



Global environment:

x 1

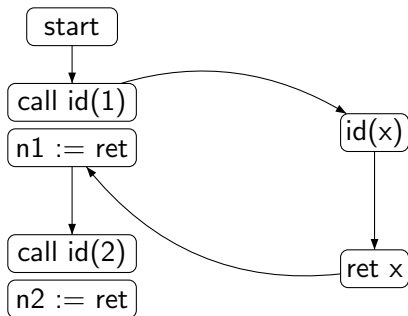
OCFA execution



Global environment:

x	1
n1	1

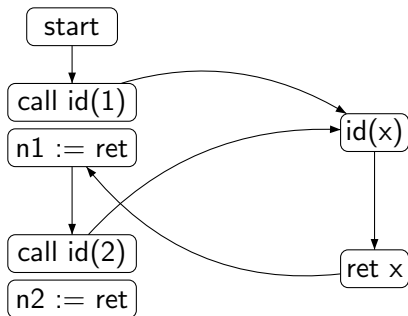
OCFA execution



Global environment:

x	1
n1	1

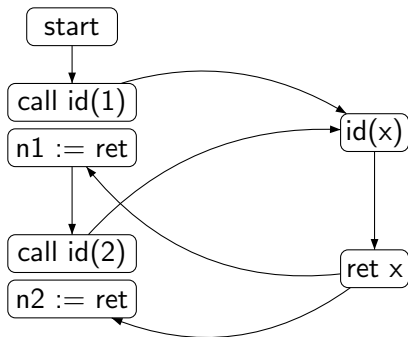
OCFA execution



Global environment:

x	1	2
n1	1	

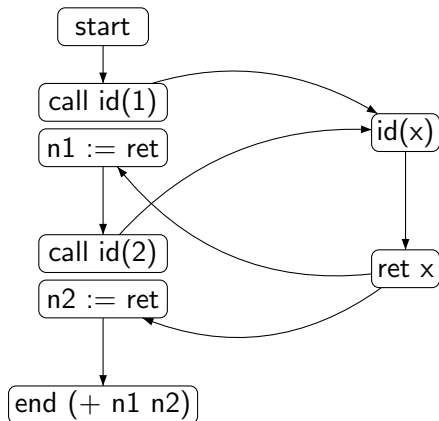
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

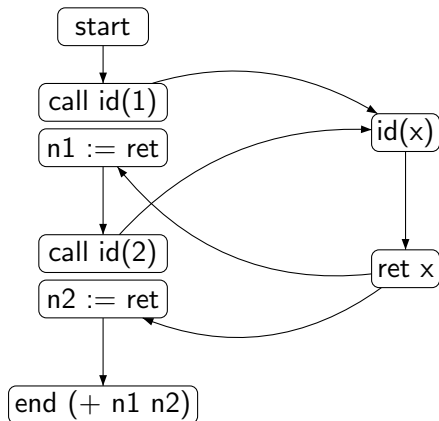
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

OCFA execution

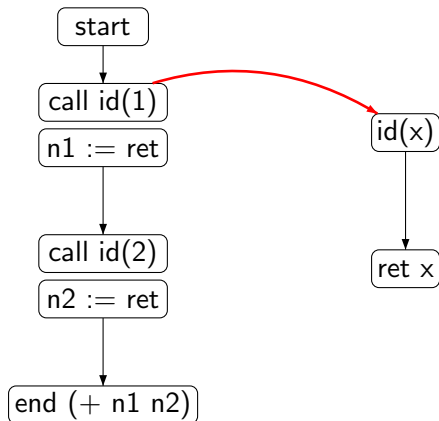


Global environment:

x	1	2
n1	1	2
n2	1	2

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

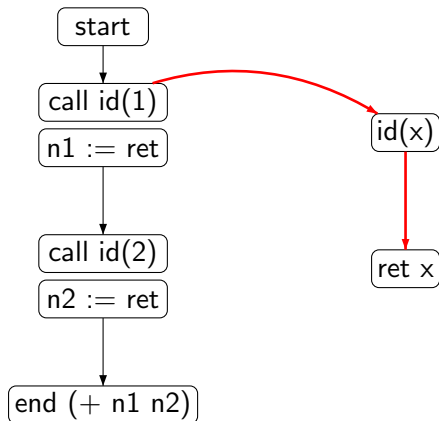
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

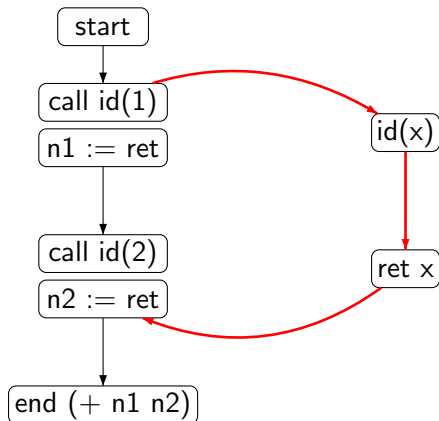
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

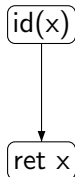
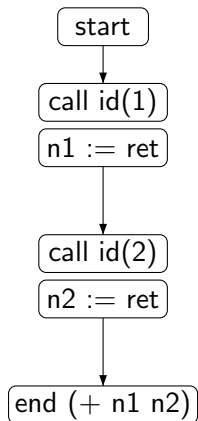
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

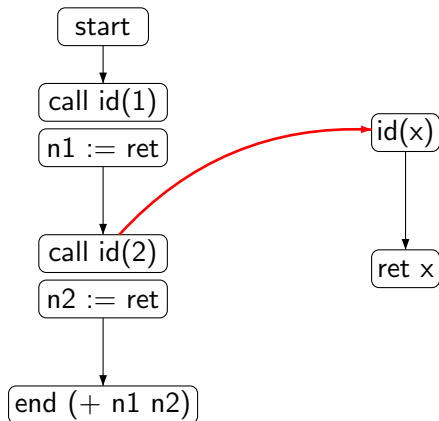
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

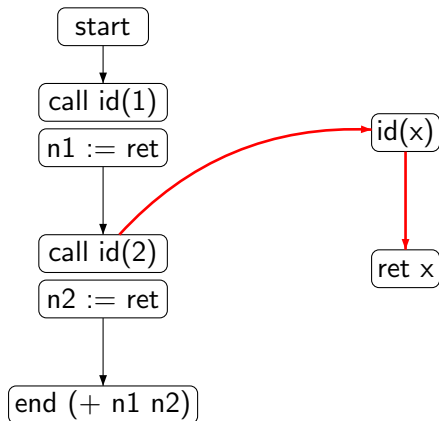
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

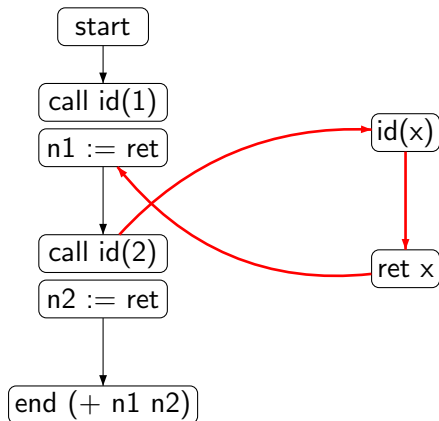
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

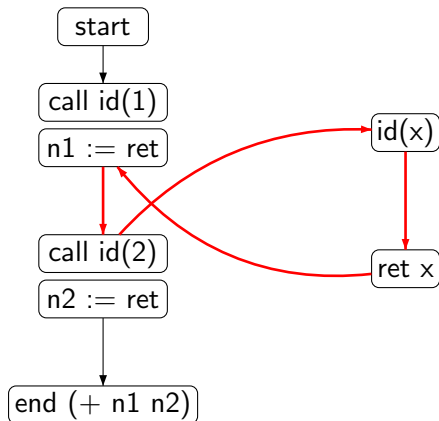
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

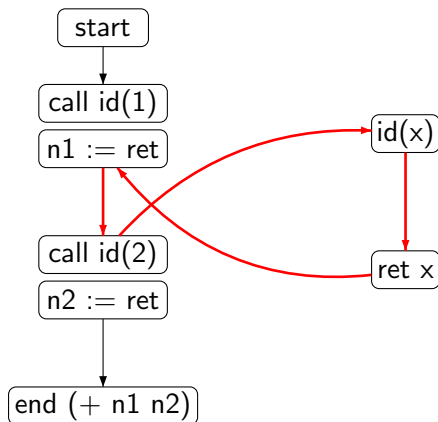
OCFA execution



Global environment:

x	1	2
n1	1	2
n2	1	2

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

Fake Rebinding

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

λ_a

λ_b

Fake Rebinding

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

Flows:

```
2(f 1(f x))
```

Fake Rebinding

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

The diagram shows two labels, λ_a and λ_b , positioned to the right of the code. Two curved arrows originate from these labels and point to the `f` parameter in the function definition. The arrow from λ_a points to the `f` in the inner function call `(f x)`, and the arrow from λ_b points to the `f` in the outer function call `(f (f x))`.

Flows:

```
2(f 1( $\lambda_a$  x))
```

Fake Rebinding

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

The diagram shows two labels, λ_a and λ_b , positioned to the right of the code. Two curved arrows originate from these labels and point to the `f` parameter in the function definition. The arrow from λ_a points to the `f` in the inner function call `(f x)`, and the arrow from λ_b points to the `f` in the outer function call `(f (f x))`.

Flows:

λ_a ¹(λ_a x) ✓

Fake Rebinding

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

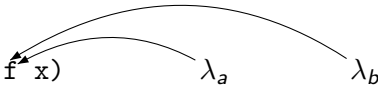
The diagram shows two lambda expressions, λ_a and λ_b , positioned to the right of the function definition. Two curved arrows originate from λ_a and λ_b and both point to the parameter f in the function definition. This illustrates that both lambda expressions refer to the same f parameter, despite the function being defined with a single f .

Flows:

$^2(\lambda_a \ ^1(\lambda_a \ x))$ ✓

$^2(\lambda_b \ ^1(\lambda_b \ x))$ ✓

Fake Rebinding

(define (compose-same f x) 
 $^2(f \ ^1(f \ x))$)

The diagram shows two lambda expressions, λ_a and λ_b , with arrows pointing to the variable f in the function definition. λ_a is positioned above the first occurrence of f , and λ_b is positioned above the second occurrence of f . The arrows indicate that both lambda expressions refer to the same variable f .

Flows:

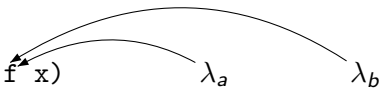
$^2(\lambda_a \ ^1(\lambda_a \ x))$ ✓

$^2(\lambda_b \ ^1(\lambda_b \ x))$ ✓

$^2(\lambda_b \ ^1(\lambda_a \ x))$ ✗

Fake Rebinding

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



Flows:

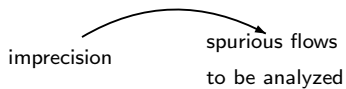
$^2(\lambda_a \ ^1(\lambda_a \ x))$ ✓

$^2(\lambda_b \ ^1(\lambda_b \ x))$ ✓

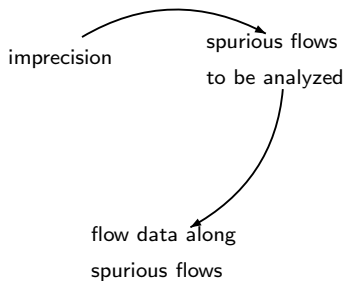
$^2(\lambda_b \ ^1(\lambda_a \ x))$ ✗

$^2(\lambda_a \ ^1(\lambda_b \ x))$ ✗

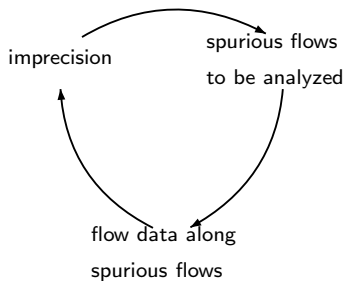
The vicious cycle of approximation



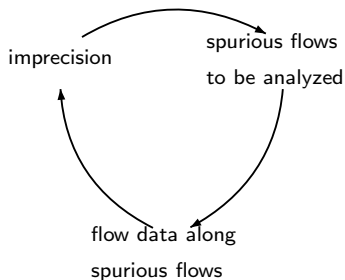
The vicious cycle of approximation



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

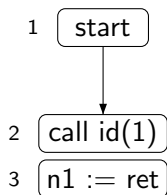
1

start

Stack:



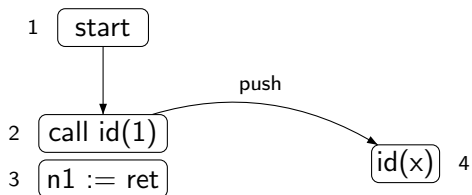
CFA2: pushdown automaton



Stack:



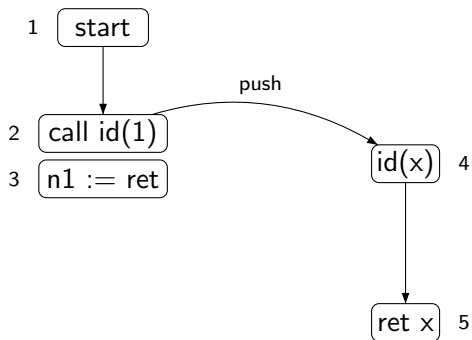
CFA2: pushdown automaton



Stack:

$x \mapsto 1, \text{ret} \mapsto 3$

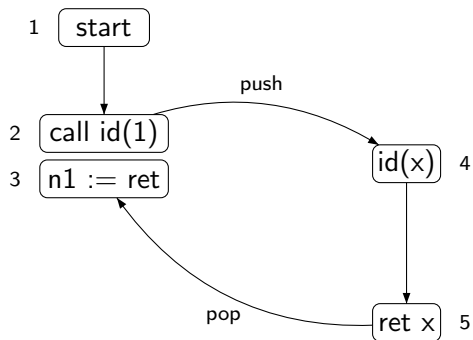
CFA2: pushdown automaton



Stack:

`x ↦ 1, ret ↦ 3`

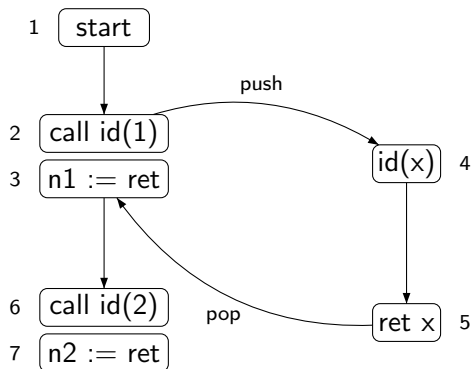
CFA2: pushdown automaton



Stack:



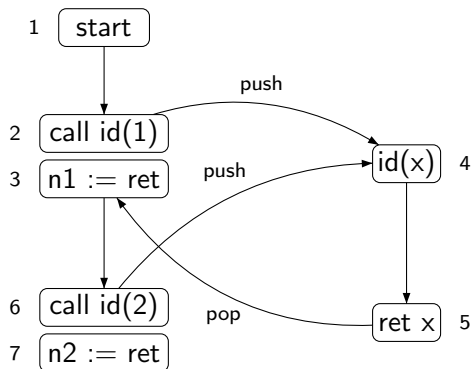
CFA2: pushdown automaton



Stack:



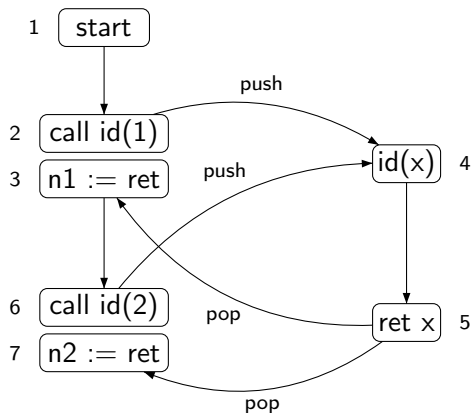
CFA2: pushdown automaton



Stack:

$x \mapsto 2, \text{ret} \mapsto 7$

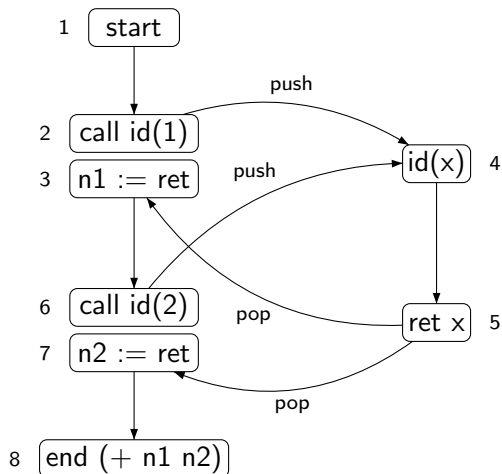
CFA2: pushdown automaton



Stack:



CFA2: pushdown automaton



Stack:



Summarization

Why:

Stack can grow arbitrarily—infinite state space

⇒ simple analysis techniques won't terminate!

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

Summarization

Why:

Stack can grow arbitrarily—infinite state space
⇒ 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.

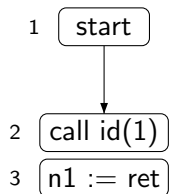
To avoid reanalyzing functions often, record summaries from function entries to function exits.

CFA2: summarization

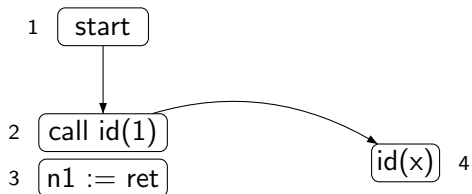
1

start

CFA2: summarization

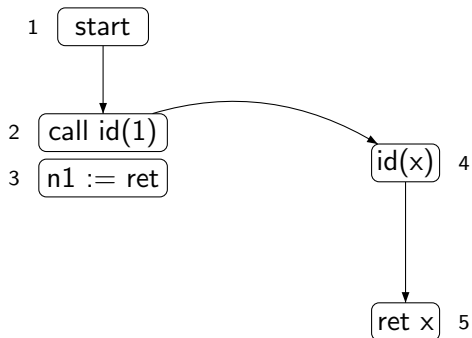


CFA2: summarization



Callers:
2 calls 4[x ↦ 1]

CFA2: summarization



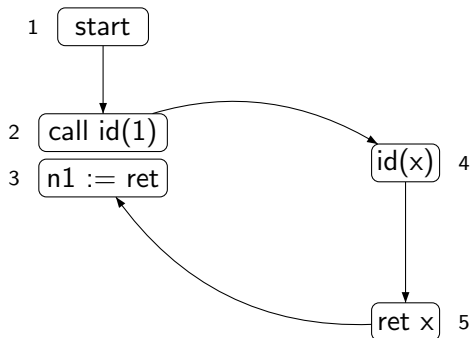
Callers:

2 calls 4[$x \mapsto 1$]

Summaries:

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

CFA2: summarization



Callers:

2 calls 4[x ↦ 1]

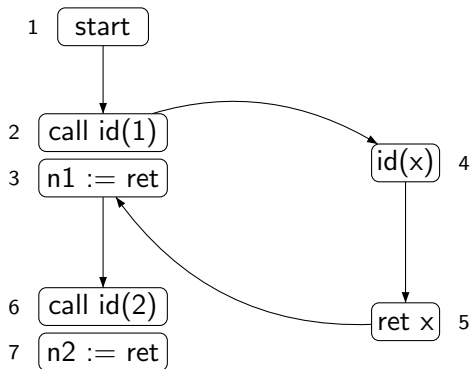
Summaries:

4[x ↦ 1] goes to 5[x ↦ 1]

Toplevel:

n1 1

CFA2: summarization



Callers:

2 calls 4[$x \mapsto 1$]

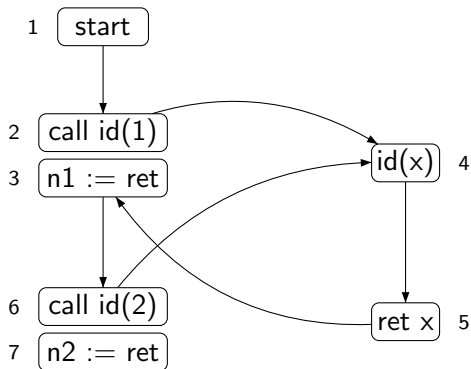
Summaries:

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

Toplevel:

n1 1

CFA2: summarization



Callers:

2 calls 4[x ↦ 1]

6 calls 4[x ↦ 2]

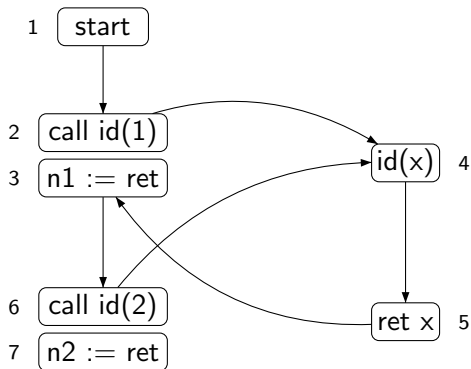
Summaries:

4[x ↦ 1] goes to 5[x ↦ 1]

Toplevel:

n1 1

CFA2: summarization



Callers:

2 calls 4[x ↦ 1]

6 calls 4[x ↦ 2]

Summaries:

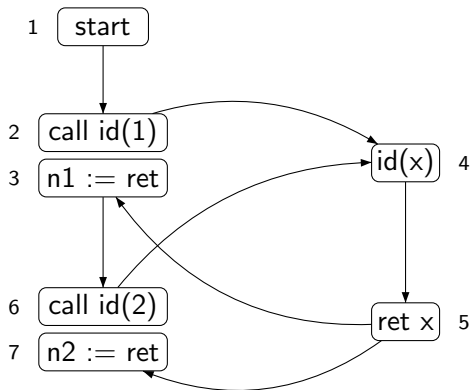
4[x ↦ 1] goes to 5[x ↦ 1]

4[x ↦ 2] goes to 5[x ↦ 2]

Toplevel:

n1 1

CFA2: summarization



Callers:

2 calls 4[x ↦ 1]

6 calls 4[x ↦ 2]

Summaries:

4[x ↦ 1] goes to 5[x ↦ 1]

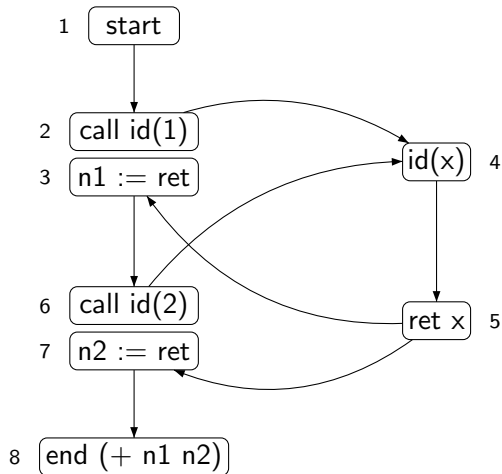
4[x ↦ 2] goes to 5[x ↦ 2]

Toplevel:

n1 1

n2 2

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)  
  2(f 1(f x)))
```

λ_a

λ_b

Stack filtering

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

The diagram shows two lambda abstractions, λ_a and λ_b , positioned above the code. Two curved arrows originate from λ_a and λ_b and point to the variable `f` in the code. The arrow from λ_a is lower and more direct, while the arrow from λ_b is higher and more curved, indicating a flow of information from the lambda expressions to the function variable `f`.

Flows:

`2(f 1(f x))`

Frame:

`[f \mapsto { λ_a , λ_b }]`

Action:

pick a lambda

Stack filtering

(define (compose-same f x)
 $^2(f \ ^1(f \ x))$)

The diagram shows two labels, λ_a and λ_b , positioned to the right of the code. Two curved arrows originate from these labels and point to the argument x in the expression $(f \ ^1(f \ x))$. The arrow from λ_a is lower and ends at the x of the inner function call, while the arrow from λ_b is higher and ends at the x of the outer function call.

Flows:

$^2(f \ ^1(f \ x))$
 $^2(f \ ^1(\lambda_a \ x))$
 $^2(\lambda_a \ ^1(\lambda_a \ x))$

Frame:

$[f \mapsto \{\lambda_a, \lambda_b\}]$
 $[f \mapsto \{\lambda_a\}]$
 $[f \mapsto \{\lambda_a\}]$

Action:

pick a lambda
commit to λ_a

Stack filtering

(define (compose-same f x)
 $^2(f \ ^1(f \ x))$)

The diagram shows two labels, λ_a and λ_b , positioned to the right of the code. Two curved arrows originate from these labels and point to the variable f in the function definition. The arrow from λ_a points to the f in the inner function call $^1(f \ x)$, and the arrow from λ_b points to the f in the outer function call $^2(f \ ^1(f \ x))$.

Flows:

$^2(f \ ^1(f \ x))$
 $^2(f \ ^1(\lambda_a \ x))$
 $^2(\lambda_a \ ^1(\lambda_a \ x))$

Frame:

$[f \mapsto \{\lambda_a, \lambda_b\}]$
 $[f \mapsto \{\lambda_a\}]$
 $[f \mapsto \{\lambda_a\}]$

Action:

pick a lambda
commit to λ_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., $(\lambda(x)(\lambda(y) (+ x y)))$

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 Γ CFA).
- ▶ 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!