Pushdown Flow Analysis of First-Class Control

Dimitris Vardoulakis   Olin Shivers

Northeastern University

Flow analysis is instrumental in building good software.

Optimization

Debugging

Verification

Development

Flow analysis is instrumental in building good software.

Optimization

Debugging

Verification

Development

What are currently the options for higher-order flow analysis?

Finite-state models

k-CFA [Shivers 91] and successors.

Approximate a program as a finite-state machine.
Call/return mismatch.

Finite-state models

k-CFA [Shivers 91] and successors.

Approximate a program as a finite-state machine.
Call/return mismatch.

But in a higher-order language, like Scheme or JavaScript, call/return is the *fundamental* control-flow mechanism.

CFA2 [ESOP 10]

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

CFA2 [ESOP 10]

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 [ESOP 10]

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.

Scheme implementation

- More precise than k-CFA
- Usually smaller state space

Summarization + first-class control $\neq \heartsuit$

Stack size is unbounded.
Summarization gets around the infinite state-space.
But requires proper nesting of calls and returns.

Summarization + first-class control $\neq \heartsuit$

Stack size is unbounded.
Summarization gets around the infinite state-space.
But requires proper nesting of calls and returns.

Many constructs break call/return nesting:

- Generators (JavaScript, Python)
- Coroutines (Lua, Simula67)
- First-class continuations (Scheme, SML/NJ, Scala)

Finite-state models

✗ Call/return mismatch, many spurious flows

✓ First-class control

Pushdown models

✓ Call/return matching, precise

✗ No first-class control

Finite-state models

✗ Call/return mismatch, many spurious flows
✓ First-class control

Pushdown models

✓ Call/return matching, precise
✗ No first-class control

Our contribution

✓ Call/return matching, precise
✓ First-class control

Overview

Background on pushdown models

Restricted continuation-passing style (RCPS)

Abstract semantics for RCPS

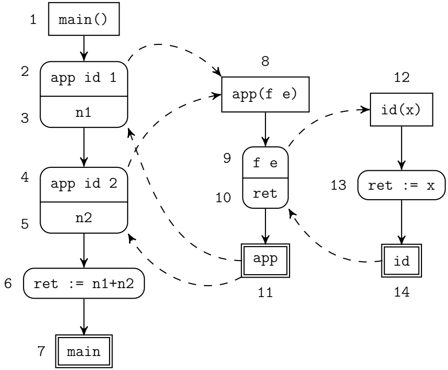
Generalizing summarization

Why pushdown models?

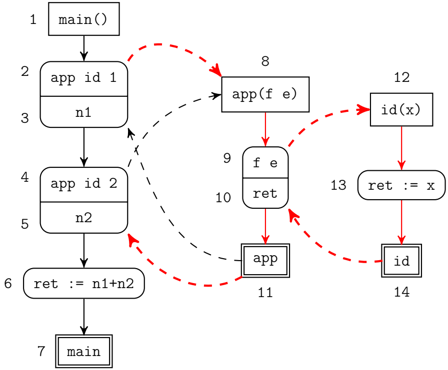
```
(define app (λ (f e) (f e)))
(define id (λ (x) x))

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

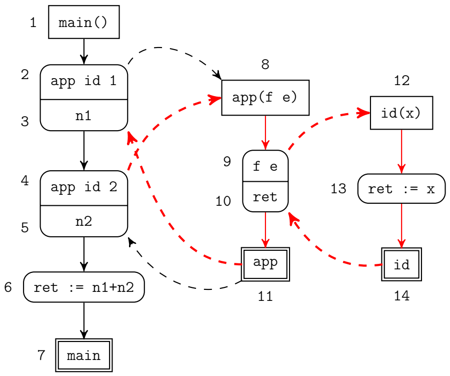
Why pushdown models?



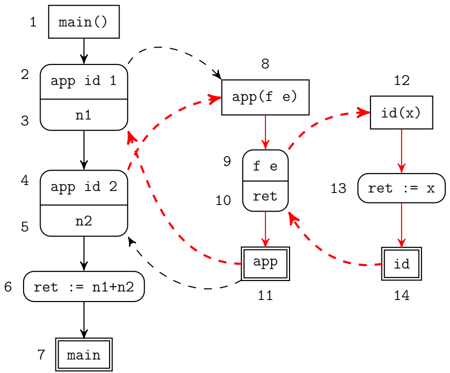
Why pushdown models?



Why pushdown models?

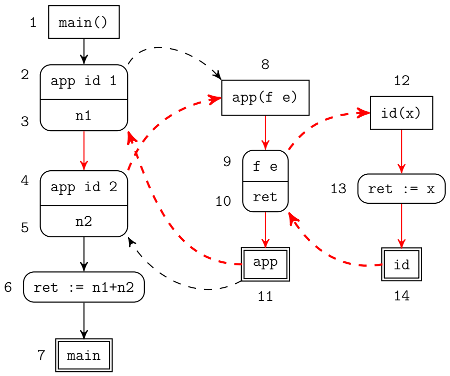


Why pushdown models?

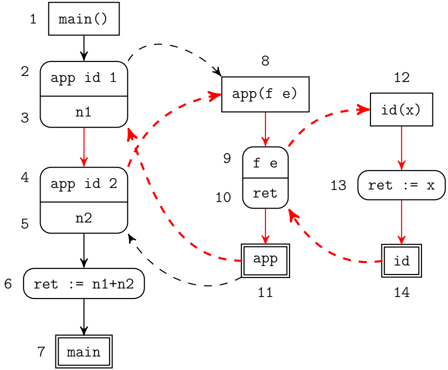


Call/return mismatch causes spurious flow of data
⇒ commonly called functions pollute the analysis.

Why pushdown models?



Why pushdown models?



Call/return mismatch causes spurious control flow
$\Rightarrow$ cannot accurately calculate stack change.

Summarization

```
    (define (g x)
        ⋮  )

    (define (f y)
        ⋮
L1:  (g 5 -3)
        ⋮
L2:  (g "a" 2)
        ⋮
L3:  (g 12 7)
        ⋮  )
```

Summarization

```
    (define (g x)
         ⋮  )

    (define (f y)
         ⋮
L1:   (g 5 -3)
         ⋮
L2:   (g "a" 2)
         ⋮
L3:   (g 12 7)
         ⋮  )
```

The computation in g doesn't depend on the call site.

Summarization

```
    (define (g x)
         ⋮  )

    (define (f y)
         ⋮
L1:   (g 5 -3)
         ⋮
L2:   (g "a" 2)
         ⋮
L3:   (g 12 7)
         ⋮  )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num)	

Summarization

```
    (define (g x)
        ⋮  )

    (define (f y)
        ⋮
L1:   (g 5 -3)
        ⋮
L2:   (g "a" 2)
        ⋮
L3:   (g 12 7)
        ⋮  )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num)	(g, Num x Num, Num)

Summarization

```
    (define (g x)
         ⋮   )

    (define (f y)
         ⋮
L1:   (g 5 -3)
         ⋮
L2:   (g "a" 2)
         ⋮
L3:   (g 12 7)
         ⋮   )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num) (g, L2, Num x Str)	(g, Num x Num, Num)

Summarization

```
    (define (g x)
        ⋮  )

    (define (f y)
        ⋮
L1:   (g 5 -3)
        ⋮
L2:   (g "a" 2)
        ⋮
L3:   (g 12 7)
        ⋮  )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num)	(g, Num x Num, Num)
(g, L2, Num x Str)	(g, Num x Str, Str)

Summarization

```
    (define (g x)
         ⋮  )

    (define (f y)
         ⋮
L1:   (g 5 -3)
         ⋮
L2:   (g "a" 2)
         ⋮
L3:   (g 12 7)
         ⋮  )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num)	(g, Num x Num, Num)
(g, L2, Num x Str)	(g, Num x Str, Str)
(g, L3, Num x Num)	

Summarization

```
    (define (g x)
        ⋮  )

    (define (f y)
        ⋮
L1:   (g 5 -3)
        ⋮
L2:   (g "a" 2)
        ⋮
L3:   (g 12 7)
        ⋮  )
```

The computation in g doesn't depend on the call site.

Callers	Summaries
(g, L1, Num x Num)	(g, Num x Num, Num)
(g, L2, Num x Str)	(g, Num x Str, Str)
(g, L3, Num x Num)	

What if g calls an escaped continuation?

The target may not even be on the stack.

Continuation-passing style

Each term is either a user or a continuation term.

```
(define (fact n k)
  (if (= n 0)
      (k 1)
      (fact (- n 1) (λ (ans) (k (* n ans))))))
```

Escaping continuations in CPS

Continuations captured in user closures may escape.

Escaping continuations in CPS

Continuations captured in user closures may escape.

```
(λ1 (f k1) (f (λ2 (u k2) (k1 u)) k1))        ;; call/cc
(λ1 (f k1) (k1 (λ2 (u k2) (f u k1))))
```

Escaping continuations in CPS

Continuations captured in user closures may escape.

```
(λ1 (f k1) (f (λ2 (u k2) (k1 u)) k1))       ;; call/cc
(λ1 (f k1) (k1 (λ2 (u k2) (f u k1))))
```

Manage CPS with a stack [Kranz et al. 86, Orbit].
Stack change from birth to use can be arbitrary.

Restricted CPS [PEPM 11]

Def: a continuation variable can appear free in a user lambda in operator position only.

Restricted CPS [PEPM 11]

Def: a continuation variable can appear free in a user lambda in operator position only.

✓ `(λ(f k1) (f (λ(u k2) (k1 u)) k1))`

✗ `(λ(f k1) (k1 (λ(u k2) (f u k1))))`

Restricted CPS [PEPM 11]

Def: a continuation variable can appear free in a user lambda in operator position only.

```
✓ (λ(f k1) (f (λ(u k2) (k1 u)) k1))
✗ (λ(f k1) (k1 (λ(u k2) (f u k1))))
✓ (λ(f k1) (k1 (λ(u k2) (f u (λ(v) (k1 v))))))
```

Restricted CPS [PEPM 11]

Def: a continuation variable can appear free in a user lambda in operator position only.

```
✓ (λ(f k1) (f (λ(u k2) (k1 u)) k1))
✗ (λ(f k1) (k1 (λ(u k2) (f u k1))))
✓ (λ(f k1) (k1 (λ(u k2) (f u (λ(v) (k1 v))))))
```

Can prove that continuation arguments live on the stack.
Force arbitrary stack change to happen only at continuation calls.

Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS λ-calculus.

Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS λ-calculus.

Concrete semantics Actual program behavior

Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS λ-calculus.

Concrete semantics	Actual program behavior
$\Downarrow$	
Abstract semantics	Reminiscent of a PDA, infinite state space

Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS λ-calculus.

Concrete semantics	Actual program behavior
$\Downarrow$	
Abstract semantics	Reminiscent of a PDA, infinite state space
$\Downarrow$	
Local semantics + summarization	No stack, finite state space Weaves calls and returns together

Semantics for escaping continuations

Control enters a user function:

$$(\llbracket (\lambda_l (u\, k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \leadsto (call, st', h')$$

$$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$$

$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$$

Semantics for escaping continuations

Control enters a user function:

$$(\llbracket (\lambda_l (u\,k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, st', h')$$

$$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$$

$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$$

Semantics for escaping continuations

Control enters a user function:

$$(\llbracket (\lambda_l (u\, k)\ call) \rrbracket, \hat{d}, \hat{c}, st, h) \leadsto (call, st', h')$$

$$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$$

$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$$

Semantics for escaping continuations

Control enters a user function:

$$(\llbracket(\lambda_l\,(u\,k)\;call)\rrbracket, \hat{d}, \hat{c}, st, h) \rightsquigarrow (call, st', h')$$

$$st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st)$$

$$h'(v) = \begin{cases} h(u) \cup \hat{d} & (v = u) \wedge H_?(u) \\ h(k) \cup \{(\hat{c}, st)\} & (v = k) \wedge H_?(k) \\ h(v) & o/w \end{cases}$$

Semantics for escaping continuations

Calling a continuation:

$$(\llbracket (q\ e)^{\gamma} \rrbracket, st, h) \rightsquigarrow (\hat{c}, \hat{d}, st', h)$$

$$\hat{d} = \hat{\mathcal{A}}_u(e, \gamma, st, h)$$

$$(\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}$$

Semantics for escaping continuations

Calling a continuation:

$$([\![(q\ e)^{\gamma}]\!], st, h) \rightsquigarrow (\hat{c}, \hat{d}, st', h)$$

$$\hat{d} = \hat{\mathcal{A}}_u(e, \gamma, st, h)$$

$$(\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}$$

Semantics for escaping continuations

Calling a continuation:

$$(\llbracket (q\ e)^{\gamma} \rrbracket, st, h) \rightsquigarrow (\hat{c}, \hat{d}, st', h)$$

$$\hat{d} = \hat{\mathcal{A}}_u(e, \gamma, st, h)$$

$$(\hat{c}, st') \in \begin{cases} \{(q, st)\} & Lam_?(q) \\ \{(st(q), pop(st))\} & S_?(\gamma, q) \\ h(q) & H_?(\gamma, q) \end{cases}$$

Summarization for RCPS

```
(λ1 (x k1) ...(λ2 (y k2) ...(k1 e) ...)  ...)
```

Summarization for RCPS

```
(λ1 (x k1) ...(λ2 (y k2) ...(k1 e) ...)  ...)
```

Traditional summaries: from the entry of λ_2 to `(k1 e)`.

Summarization for RCPS

`(λ1 (x k1) ...(λ2 (y k2) ...(k1 e) ...)  ...)`

Traditional summaries: from the entry of λ_2 to `(k1 e)`.

Instead, record entries of λ_1 as we see them.
Create cross-procedure summaries from λ_1 entries to `(k1 e)`.

Summarization for RCPS

```
(λ₁ (x k1) ...(λ₂ (y k2) ...(k1 e) ...)  ...)
```

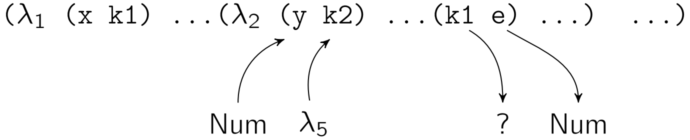
Summarization for RCPS

(λ_1 `(x k1) ...`(λ_2 `(y k2) ...(k1 e) ...)  ...)`

Num $\quad \lambda_5$

Callers: (λ_2, λ_5, Num)
Summaries:

Summarization for RCPS



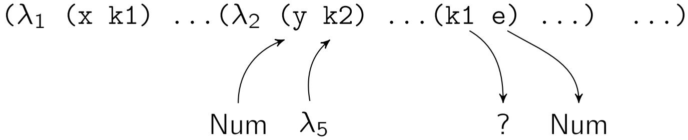
Callers: (λ_2, λ_5, Num)
Summaries:

Summarization for RCPS

(λ_1 `(x k1) ...(`λ_2 `(y k2) ...(k1 e) ...)  ...)`

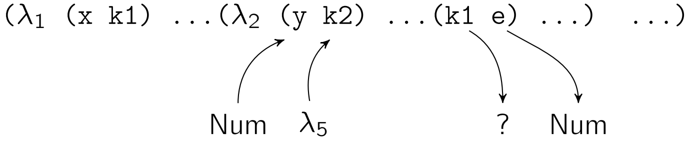
Callers: (λ_2, λ_5, Num), (λ_1, λ_4, Str), (λ_1, λ_7, Bool)
Summaries:

Summarization for RCPS



Callers: (λ_2, λ_5, Num), (λ_1, λ_4, Str), (λ_1, λ_7, Bool)
Summaries: (λ_1, Str, Num), (λ_1, Bool, Num)

Summarization for RCPS



Callers: $(\lambda_2, \lambda_5, \text{Num})$, $(\lambda_1, \lambda_4, \text{Str})$, $(\lambda_1, \lambda_7, \text{Bool})$
Summaries: $(\lambda_1, \text{Str}, \text{Num})$, $(\lambda_1, \text{Bool}, \text{Num})$

Conclusions

Pushdown analyses model call/return faithfully.
Fewer spurious control and data flows.

Conclusions

Pushdown analyses model call/return faithfully.
Fewer spurious control and data flows.

In Restricted CPS continuations escape in a well-behaved way.

Handle escaping continuations by generalizing summaries.

Conclusions

Pushdown analyses model call/return faithfully.
Fewer spurious control and data flows.

In Restricted CPS continuations escape in a well-behaved way.

Handle escaping continuations by generalizing summaries.

CFA2 a drop-in replacement of k-CFA.

Conclusions

Pushdown analyses model call/return faithfully.
Fewer spurious control and data flows.

In Restricted CPS continuations escape in a well-behaved way.

Handle escaping continuations by generalizing summaries.

CFA2 a drop-in replacement of k-CFA.

Thank you!