Pushdown Flow Analysis of First-Class Control

Dimitris Vardoulakis    Olin Shivers

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

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

```plaintext
main()
1
app id 1
2
n1
3
app id 2
4
n2
5
ret := n1+n2
6
main
7
app(f e)
8
f e
9
ret
10
app(id(x))
11
id(x)
12
ret := x
13
id
14
```

Diagram:

1. `main()`
2. `app id 1`
   - `n1`
3. `app id 2`
   - `n2`
4. `ret := n1+n2`
5. `main`
6. `app(f e)`
7. `f e`
8. `ret`
9. `app(id(x))`
10. `id(x)`
11. `ret := x`
12. `id`
Why pushdown models?

```
main()

app id 1
n1

app id 2
n2

ret := n1 + n2

app(f e)
f e
ret

app(id(x))

id(x)

id

ret := x
```
Why pushdown models?

```plaintext
1. main()
2. app id 1
   n1
3. app id 2
   n2
4. ret := n1 + n2
5. main
6. app(f e)
7. app
   ret
8. id(x)
9. ret := x
10. id
```
Why pushdown models?

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

main()

app id 1
n1

app id 2
n2

ret := n1+n2

app(f e)

app(id(x))

f e
ret

id(x)

ret := x

id

app

main
Why pushdown models?

Call/return mismatch causes spurious control flow
⇒ cannot accurately calculate stack change.
Summarization

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

The computation in \texttt{g} doesn't depend on the call site. Callers Summaries \hspace{1cm} \texttt{(g, L1, Num x Num)} \hspace{1cm} \texttt{(g, Num x Num, Num)} \hspace{1cm} \texttt{(g, L2, Num x Str)} \hspace{1cm} \texttt{(g, Num x Str, Str)} \hspace{1cm} \texttt{(g, L3, Num x Num)}

What if \texttt{g} calls an escaped continuation? The target may not even be on the stack.
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.
(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)
(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.

<table>
<thead>
<tr>
<th>Callers</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g, L1, Num x Num)</td>
<td>(g, Num x Num, Num)</td>
</tr>
</tbody>
</table>

Summarization

\begin{align*}
&\text{(define (g x)} \\
&\quad : )
\end{align*}

\begin{align*}
&\text{(define (f y)} \\
&\quad : )
\end{align*}

\begin{align*}
L1: &\quad (g \ 5 \ -3) \\
L2: &\quad (g \ "a" \ 2) \\
L3: &\quad (g \ 12 \ 7) \\
\end{align*}

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

\begin{tabular}{l|l}
\text{Callers} & \text{Summaries} \\
\hline
(g, L1, Num x Num) & (g, Num x Num, Num) \\
(g, L2, Num x Str) & \\
\end{tabular}

What if g calls an escaped continuation? The target may not even be on the stack.
(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.

<table>
<thead>
<tr>
<th>Callers</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g, L1, Num x Num)</td>
<td>(g, Num x Num, Num)</td>
</tr>
<tr>
<td>(g, L2, Num x Str)</td>
<td>(g, Num x Str, Str)</td>
</tr>
</tbody>
</table>
(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.

<table>
<thead>
<tr>
<th>Callers</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g, L1, Num x Num)</td>
<td>(g, Num x Num, Num)</td>
</tr>
<tr>
<td>(g, L2, Num x Str)</td>
<td>(g, Num x Str, Str)</td>
</tr>
<tr>
<td>(g, L3, Num x Num)</td>
<td></td>
</tr>
</tbody>
</table>
(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

<table>
<thead>
<tr>
<th>Callers</th>
<th>Summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>(g, L1, Num x Num)</td>
<td>(g, Num x Num, Num)</td>
</tr>
<tr>
<td>(g, L2, Num x Str)</td>
<td>(g, Num x Str, Str)</td>
</tr>
<tr>
<td>(g, L3, Num x Num)</td>
<td></td>
</tr>
</tbody>
</table>

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))))))
Continuations captured in user closures may escape.
Escaping continuations in CPS

Continuations captured in user closures may escape.

\[(\lambda_1 \ (f \ k_1) \ (f \ (\lambda_2 \ (u \ k_2) \ (k_1 \ u)) \ k_1)) \ ; \ \text{call/cc} \ \ (\lambda_1 \ (f \ k_1) \ (k_1 \ (\lambda_2 \ (u \ k_2) \ (f \ u \ k_1))))\]
Escaping continuations in CPS

Continuations captured in user closures may escape.

\[(\lambda_1 (f \; k1) (f (\lambda_2 (u \; k2) (k1 \; u)) \; k1)) \; ; \; \text{call/cc} \]
\[(\lambda_1 (f \; k1) (k1 (\lambda_2 (u \; k2) (f \; u \; k1))))\]

Manage CPS with a stack [Kranz et al. 86, Orbit]. Stack change from birth to use can be arbitrary.
Def: a continuation variable can appear free in a user lambda in operator position only.
Def: a continuation variable can appear free in a user lambda in operator position only.

✓ $(\lambda(f \; k1) \; (f \; (\lambda(u \; k2) \; (k1 \; u)) \; k1))$

✗ $(\lambda(f \; k1) \; (k1 \; (\lambda(u \; k2) \; (f \; u \; k1))))$
Def: a continuation variable can appear free in a user lambda in operator position only.

\[
\begin{align*}
\checkmark & \quad (\lambda(f \ k1) \ (f \ (\lambda(u \ k2) \ (k1 \ u)) \ k1)) \\
\times & \quad (\lambda(f \ k1) \ (k1 \ (\lambda(u \ k2) \ (f \ u \ k1)))) \\
\checkmark & \quad (\lambda(f \ k1) \ (k1 \ (\lambda(u \ k2) \ (f \ u \ (\lambda(v) \ (k1 \ v))))))
\end{align*}
\]
Restricted CPS [PEPM 11]

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

✓  \((\lambda(f \ k1) \ (f \ (\lambda(u \ k2) \ (k1 \ u)) \ k1))\)

✗  \((\lambda(f \ k1) \ (k1 \ (\lambda(u \ k2) \ (f \ u \ k1))))\)

✓  \((\lambda(f \ k1) \ (k1 \ (\lambda(u \ k2) \ (f \ u \ (\lambda(v) \ (k1 \ v)))))\))

Can prove that continuation arguments live on the stack.
Force arbitrary stack change to happen only at continuation calls.
Abstract interpretation of programs in RCPS $\lambda$-calculus.
Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS $\lambda$-calculus.

Concrete semantics  Actual program behavior
Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS $\lambda$-calculus.

Concrete semantics

$\Downarrow$

Abstract semantics

Actual program behavior

Reminiscent of a PDA, infinite state space
Theoretical formulation of CFA2

Abstract interpretation of programs in RCPS $\lambda$-calculus.

Concrete semantics $\Downarrow$ Abstract semantics $\Downarrow$ Local semantics $+$ summarization

<table>
<thead>
<tr>
<th>Concrete semantics</th>
<th>Actual program behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Downarrow$</td>
<td>Reminiscent of a PDA,</td>
</tr>
<tr>
<td>Abstract semantics</td>
<td>infinite state space</td>
</tr>
<tr>
<td>$\Downarrow$</td>
<td>No stack, finite state space</td>
</tr>
<tr>
<td>Local semantics</td>
<td>Weaves calls and returns together</td>
</tr>
</tbody>
</table>
Semantics for escaping continuations

Control enters a user function:

\[
((\lambda_1(u \ k) \ call), \hat{d}, \hat{c}, st, h) \sim (call, st', h')
\]

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

\[
h'(v) = \begin{cases} 
    h(u) \cup \hat{d} & (v = u) \land H?(u) \\
    h(k) \cup \{(\hat{c}, st)\} & (v = k) \land H?(k) \\
    h(v) & o/w
\end{cases}
\]
Semantics for escaping continuations

Control enters a user function:

\[
[[\lambda (u \ k) \ \text{call}]], \hat{d}, \hat{c}, st, h) \leadsto (\text{call}, st', h')
\]

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

\[
h'(v) = \begin{cases} 
h(u) \cup \hat{d} & (v = u) \land H?(u) \\
h(k) \cup \{(\hat{c}, st)\} & (v = k) \land H?(k) \\
h(v) & o/w \end{cases}
\]
Control enters a user function:

\[
\begin{align*}
&([\lambda I(u \, k) \, \text{call}], \hat{d}, \hat{c}, st, h) \sim (\text{call}, st', h') \\
&st' = push([u \mapsto \hat{d}][k \mapsto \hat{c}], st) \\
&h'(v) = \begin{cases} 
  h(u) \cup \hat{d} & (v = u) \land H?(u) \\
  h(k) \cup \{(\hat{c}, st)\} & (v = k) \land H?(k) \\
  h(v) & \text{otherwise}
\end{cases}
\end{align*}
\]
Semantics for escaping continuations

Control enters a user function:

\[
([\lambda_l(u k) \text{call}]), \hat{d}, \hat{c}, st, h) \leadsto (\text{call}, st', h')
\]

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

\[
h'(v) = \begin{cases}
  h(u) \cup \hat{d} & (v = u) \land H? (u) \\
  h(k) \cup \{(\hat{c}, st)\} & (v = k) \land H? (k) \\
  h(v) & \text{o/w}
\end{cases}
\]
Semantics for escaping continuations

Calling a continuation:

\[
\begin{align*}
\& [(q \ e)^\gamma], \ st, \ h) \rightarrow (\hat{c}, \hat{d}, \ st', \ h) \\
\& \hat{d} = \hat{A}_u(e, \gamma, \ st, \ h) \\
\& (\hat{c}, \ st') \in \begin{cases} 
\{(q, st)\} & Lam? (q) \\
\{(st(q), \ pop(st))\} & S? (\gamma, q) \\
h(q) & H? (\gamma, q) 
\end{cases}
\end{align*}
\]
Semantics for escaping continuations

Calling a continuation:

\[
(\llbracket (q \ e)^{\gamma} \rrbracket, st, h) \leadsto (\hat{c}, \hat{d}, st', h) \\
\hat{d} = \hat{\lambda}_u(e, \gamma, st, h) \\
(\hat{c}, st') \in \begin{cases} 
\{(q, st)\} & \text{Lam?}(q) \\
\{(st(q), \text{pop}(st))\} & \text{S?}(\gamma, q) \\
h(q) & \text{H?}(\gamma, q) 
\end{cases}
\]
Semantics for escaping continuations

Calling a continuation:

\[
\left[ (q \ e)^\gamma \right], \ st, \ h \xrightarrow{\sim} (\hat{c}, \hat{d}, \ st', \ h) \\
\hat{d} = \hat{A}_u(e, \gamma, \ st, \ h) \\
\begin{cases} 
((q, \ st)) & \text{Lam}_? (q) \\
((st(q), \ \text{pop}(st))) & \text{S}_? (\gamma, \ q) \\
h(q) & \text{H}_? (\gamma, \ q)
\end{cases}
\]
Summarization for RCPS

\[(\lambda_1 (x \; k1) \ldots (\lambda_2 (y \; k2) \ldots (k1 \; e) \ldots) \ldots) \ldots\]
Summarization for RCPS

\[(\lambda_1 (x \, k_1) \ldots (\lambda_2 (y \, k_2) \ldots (k_1 \, e) \ldots) \ldots) \ldots\]

Traditional summaries: from the entry of \(\lambda_2\) to \((k_1 \, e)\).
Summarization for RCPS

\[(\lambda_1 (x \ k1) \ldots (\lambda_2 (y \ k2) \ldots (k1 \ e) \ldots) \ldots) \ldots\]

Traditional summaries: from the entry of \(\lambda_2\) to \((k1 \ e)\).

Instead, record entries of \(\lambda_1\) as we see them.
Create cross-procedure summaries from \(\lambda_1\) entries to \((k1 \ e)\).
Summarization for RCPS

\[(\lambda_1 (x \text{ } k1) \ldots (\lambda_2 (y \text{ } k2) \ldots (k1 \text{ } e) \ldots) \ldots) \ldots\]
Summarization for RCPS

$$(\lambda_1 (x \; k1) \ldots (\lambda_2 (y \; k2) \ldots (k1 \; e) \ldots) \ldots) \ldots)$$

Callers: $(\lambda_2, \; \lambda_5, \; \text{Num})$

Summaries:
Summarization for RCPS

$$(\lambda_1 (x \ k1) \ldots (\lambda_2 (y \ k2) \ldots (k1 \ e) \ldots) \ldots)$$

Callers: $$(\lambda_2, \lambda_5, \text{Num})$$
Summaries:
Summarization for RCPS

\[(\lambda_1 (x \ k1) \ldots (\lambda_2 (y \ k2) \ldots (k1 \ e) \ldots) \ldots) \ldots\]

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

Summaries:
Summarization for RCPS

\[(\lambda_1 (x \ k1) \ldots (\lambda_2 (y \ k2) \ldots (k1 \ e) \ldots)) \ldots)\]

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

\[(\lambda_1 (x \ k1) \ldots (\lambda_2 (y \ k2) \ldots (k1 \ e) \ldots) \ldots) \ldots)\]

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!