Ordering Multiple Continuations on the Stack

Dimitrios Vardoulakis Olin Shivers

Northeastern University

CPS in practice

CPS widely used in functional-language compilation.

Multiple continuations (conditionals, exceptions, *etc*).

Use a stack to manage them.

Contributions

- Syntactic restriction on multi-continuation CPS
  for better reasoning about stack.
- Static analysis for efficient multi-continuation CPS.

Overview

- Background:
  Continuation-passing style (CPS)
  Multi-continuation CPS
  CPS with a runtime stack
- Restricted CPS (RCPS)
- Continuation-age analysis
- Evaluation

Continuation-passing style (CPS)

Characteristics

- Each function takes a continuation argument, “returns” by calling it.
- All intermediate computations are named.
- Continuations reified as lambdas.

Continuation-passing style (CPS)

Characteristics

- Each function takes a continuation argument, "returns" by calling it.
- All intermediate computations are named.
- Continuations reified as lambdas.

Example

```
(define (discr a b c)
  (- (* b b) (* 4 a c)))
```

Continuation-passing style (CPS)

Characteristics

- Each function takes a continuation argument, “returns” by calling it.
- All intermediate computations are named.
- Continuations reified as lambdas.

Example

```
(define (discr a b c)
  (- (* b b) (* 4 a c)))
```

$\xrightarrow{\text{CPS}}$

```
(define (discr a b c k)
  (%* b b
      (λ(p1)
        (%* 4 a c
            (λ(p2)
              (%- p1 p2
                  (λ(d)(k d))))))))
```

Partitioned CPS [Steele 78, Rabbit]

```
(define (discr a b c k)
  (%* b b
      (λ(p1)
        (%* 4 a c
            (λ(p2)
              (%- p1 p2
                  (λ(d)(k d))))))))
```

- Variables, lambdas and calls split into disjoint sets, "user" and "continuation".
- Calls classified depending on operator.

Multi-continuation CPS

```
;; Add all positive numbers in the list
(define (add-pos l)
  (if (null? l)
    0
    (let ((fst (car l))
          (rest (cdr l)))
      (if (< 0 fst)
          (+ fst (add-pos rest))
          (add-pos rest))))))
```

Multi-continuation CPS: Conditionals

```
(define (add-pos l k)
  ...
  (%if pos-fst
       (λ() (add-pos rest
                     (λ(res) (%+ fst res k))))
       (λ() (add-pos rest k))))
```

Multi-continuation CPS: Exception handlers

```
(define (add-pos l k-ret k-exn)
  ...
  (λ(fst)
    (%number? fst
              (λ(num-fst)
                (%if num-fst
                     (λ() ...)
                     (λ() (k-exn "Not a list of numbers.")))))))
```

Compile CPS without stack [Steele 78, Rabbit]

Argument evaluation pushes stack, function calls are jumps.

In CPS, every call is a tail call.

All closures in heap.

GC pressure.

Compile CPS with a stack [Kranz 88, Orbit]

Tail calls from direct style,
continuation argument is a variable.

```
(define (add-pos l k)
  ...
  (%if pos-fst
       (λ()(add-pos rest (λ(res)(%+ fst res k))))
       (λ()(add-pos rest k))))
```

Escaping continuations

(λ_1(f k) (k (λ_2(g k2) (g 42 k))))

Escaping continuations

(λ_1(f k) (k (λ_2(g k2) (g 42 k))))

No capturing of continuation variables by user closures [Sabry-Felleisen 92], [Danvy-Lawall 92].

Restricted CPS (RCPS)

- A user lambda doesn't contain free continuation variables,
- Or it's α-equivalent to `(λ(f cc)(f (λ(x k)(cc x)) cc))`

Restricted CPS (RCPS)

- A user lambda doesn't contain free continuation variables,
- Or it's α-equivalent to `(λ(f cc)(f (λ(x k)(cc x)) cc))`

For example,

```
(λ₁(u k1 k2)(u (λ₂(k3)(k3 u))
               k1
               (λ₃(v)(k2 v))))
```

What does RCPS buy us?

Continuations escape in a controlled way.

Theorem: Continuations in argument position are stackable.

What does RCPS buy us?

Continuations escape in a controlled way.

Theorem: Continuations in argument position are stackable.
Proof?

The lifetime of a continuation argument

Doesn't escape:

```
((λ(u k) (k u))
 "foo"                  ✓
 clam)
```

The lifetime of a continuation argument

Operator, escapes:

```
((λ(u cc) (f (λ(x k) (cc x)) cc))
 (λ(v k) (k v))                                ✓
 clam)
```

The lifetime of a continuation argument

Argument, escapes:

```
((λ(k) (k (λ(u k2) (u k))))
 clam)
```

✗

Extending the Orbit stack policy

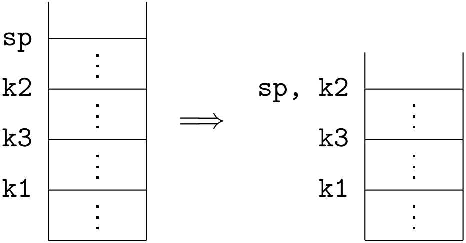
Tail calls with multiple continuations:

```
(f e1 e2 k1 k2 k3)
```

Extending the Orbit stack policy

Tail calls with multiple continuations:

`(f e1 e2 k1 k2 k3)`



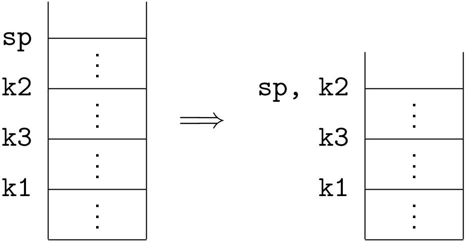
sp
k2
k3
k1

⟹

sp, k2
k3
k1

Extending the Orbit stack policy

Tail calls with multiple continuations:
`(f e1 e2 k1 k2 k3)`



In general, can't find youngest continuation statically.
At runtime, compare pointers of `k1`, `k2`, `k3` to `sp`.

Continuation-Age (Cage) analysis

Possible solution:
compare ages of continuation closures that flow to call site.

```
((λ(f k)
   ... (f "foo" clam₁ k) ...
   ... (f "bar" clam₂ clam₃) ...)
 (λ(u k1 k2) call)
 halt)
```

Continuation-Age (Cage) analysis

Possible solution:
compare ages of continuation closures that flow to call site.

```
((λ(f k)
   ... (f "foo" clam₁ k) ...
   ... (f "bar" clam₂ clam₃) ...)
 (λ(u k1 k2) call)
 halt)
```

`k1`: *clam*$_1$, *clam*$_2$
`k2`: *halt*, *clam*$_3$

Continuation-Age (Cage) analysis

Possible solution:
compare ages of continuation closures that flow to call site.

```
((λ(f k)
   ... (f "foo" clam₁ k) ...
   ... (f "bar" clam₂ clam₃) ...)
 (λ(u k1 k2) call)
 halt)
```

$clam_1 \preceq halt$ ✓

$clam_2 \preceq clam_3$ ✓

`k1`: *clam*$_1$, *clam*$_2$

`k2`: *halt*, *clam*$_3$

Continuation-Age (Cage) analysis

Possible solution:
compare ages of continuation closures that flow to call site.

```
((λ(f k)
   ... (f "foo" clam1 k) ...
   ... (f "bar" clam2 clam3) ...)
 (λ(u k1 k2) call)
 halt)
```

`k1`: $clam_1$, $clam_2$
`k2`: $halt$, $clam_3$

$clam_1 \preceq halt$ ✓
$clam_2 \preceq clam_3$ ✓
$clam_2 \preceq halt$ ✓
$clam_1 \preceq clam_3$ ✗

Cage analysis: take two

```
((λ(f k)
   ... (f "foo" clam₁ k) ...
   ... (f "bar" clam₂ clam₃) ...)
 (λ(u k1 k2) call)
 halt)
```

Better solution (possible by RCPS):

- Reason about continuation variables directly.
- Record total orders of continuation variables bound by the same user lambda.

Cage analysis: Ordering continuation variables

```
((λ(f k)
   ... (f "foo" clam₁ k) ...
   ... (f "bar" clam₂ clam₃) ...)
 (λ(u k1 k2) call)
 halt)
```

1st call $\mathtt{k1} \preceq \mathtt{k2}$

Cage analysis: Ordering continuation variables

```
((λ(f k)
   ... (f "foo" clam_1 k) ...
   ... (f "bar" clam_2 clam_3) ...)
 (λ(u k1 k2) call)
 halt)
```

1st call	$\texttt{k1} \preceq \texttt{k2}$
2nd call	$\texttt{k1} \preceq \texttt{k2}$
Overall	$\texttt{k1} \preceq \texttt{k2}$

Cage analysis: Flowing age information

$(\lambda_1$`(u1 k1 k2 k3)`
`... (u1 k1 k3` clam_2 clam_3`) ...)`

On entering λ_1:

- $\langle\{\texttt{k3}\},\{\texttt{k1}\},\{\texttt{k2}\}\rangle$
- `u1` bound to `(`λ_4`(k4 k5 k6 k7)`*call*`)`

Cage analysis: Flowing age information

```
(λ₁(u1 k1 k2 k3)
   ... (u1 k1 k3 clam₂ clam₃) ...)
```

On entering λ_1:

- $\langle \{\texttt{k3}\}, \{\texttt{k1}\}, \{\texttt{k2}\} \rangle$
- `u1` bound to `(`λ_4`(k4 k5 k6 k7)`*call*`)`

`k2` not used $\langle \{\texttt{k3}\}, \{\texttt{k1}\} \rangle$

Cage analysis: Flowing age information

$(\lambda_1(\texttt{u1 k1 k2 k3})$
$\quad \texttt{... (u1 k1 k3 } \mathit{clam}_2 \; \mathit{clam}_3\texttt{) ...)}$

On entering λ_1:

- $\langle\{\texttt{k3}\},\{\texttt{k1}\},\{\texttt{k2}\}\rangle$
- `u1` bound to $(\lambda_4(\texttt{k4 k5 k6 k7})\,\mathit{call})$

`k2` not used	$\langle\{\texttt{k3}\},\{\texttt{k1}\}\rangle$
clam_2, clam_3 new	$\langle\{\mathit{clam}_2,\mathit{clam}_3\},\{\texttt{k3}\},\{\texttt{k1}\}\rangle$

Cage analysis: Flowing age information

$(\lambda_1$`(u1 k1 k2 k3)`
`  ... (u1 k1 k3` clam_2 clam_3`) ...)`

On entering λ_1:

- $\langle\{\mathtt{k3}\},\{\mathtt{k1}\},\{\mathtt{k2}\}\rangle$
- `u1` bound to $(\lambda_4$`(k4 k5 k6 k7)`*call*)

`k2` not used	$\langle\{\mathtt{k3}\},\{\mathtt{k1}\}\rangle$
clam_2, clam_3 new	$\langle\{\mathit{clam}_2,\mathit{clam}_3\},\{\mathtt{k3}\},\{\mathtt{k1}\}\rangle$
actuals to formals	$\langle\{\mathtt{k6},\mathtt{k7}\},\{\mathtt{k5}\},\{\mathtt{k4}\}\rangle$

Also in the paper

- RCPS natural fit for multi-return lambda calculus.
- Multi-return lambda calculus $\overset{\text{CPS}}{\Longrightarrow}$ RCPS
- Implementation in Scheme48.

Evaluation

LALR parser in RCPS

184 multi-continuation calls (152 two-cont, 32 three-cont)

164 variable only

Evaluation

LALR parser in RCPS

184 multi-continuation calls (152 two-cont, 32 three-cont)
164 variable only

Cage with $k = 0$

142 resolved completely (87%)
22 resolved partially (ruled out one continuation)

Evaluation

LALR parser in RCPS

184 multi-continuation calls (152 two-cont, 32 three-cont)
164 variable only

Cage with $k = 0$

142 resolved completely (87%)
22 resolved partially (ruled out one continuation)

Control is less variant than data.

Conclusions

- Manage multi-continuation CPS with a stack.
- RCPS enables better reasoning about stack.
- Cage analysis to find youngest continuation statically.

Conclusions

- Manage multi-continuation CPS with a stack.
- RCPS enables better reasoning about stack.
- Cage analysis to find youngest continuation statically.

Thank you!