Static vs. Dynamic Delimited Continuations

Dariusz Biernacki

Institute of Computer Science
University of Wrocław

Group of Programming Languages Seminar
Institute of CS, University of Wrocław

19 October 2007
1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
1 Introduction
   ■ Functional Programming and Continuations
   ■ Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   ■ Static Delimited Continuations
   ■ Dynamic Delimited Continuations
   ■ Comparison and Issues

3 Dynamic Continuation-Passing Style
   ■ New Operational Characterization
   ■ Higher-Order Characterization
   ■ Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Outline

1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- **Example**: left-to-right evaluation of $(1 + 2) \times (3 + 4)$

```
(1 + 2) \times (3 + 4)  [ ]
1 + 2                  [ ] \times (3 + 4)
1                      \text{(1 + [ ] \times (3 + 4))}
2                      \text{(1 + [ ] \times (3 + 4))}
3                      3 \times [ ]
3 + 4                  \text{3 \times ([ ] + 4)}
3                      \text{3 \times (3 + [ ])}
4                      \text{3 \times [ ]}
7                      [ ]
21                     [ ]
```
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- **Example**: left-to-right evaluation of \((1 + 2) \times (3 + 4)\)

\[
(1 + 2) \times (3 + 4) \\
1 + 2 \\
1 + 2 \\
2 \\
2 \\
3 \\
3 + 4 \\
3 + 4 \\
3 \\
3 \\
4 \\
4 \\
7 \\
21
\]
Continuations

- Continuation: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of $(1 + 2) \times (3 + 4)$

\[
(1 + 2) \times (3 + 4) \\
1 + 2 \\
1 + 2 \\
1 + 2 \\
3 \\
3 + 4 \\
3 + 4 \\
3 + 4 \\
3 + 4 \\
3 \times [ ] \\
3 \times (3 + [ ]) \\
3 \times (3 + [ ]) \\
3 \times (3 + [ ]) \\
3 \times [ ] \\
21
\]
**Continuations**

- **Continuation**: the “rest of the computation” at a given point of a program execution

- **Example**: left-to-right evaluation of $(1 + 2) \times (3 + 4)$

```
(1 + 2) * (3 + 4)  []
  1 + 2          [ ] * (3 + 4)
    1             ([ ] + 2) * (3 + 4)
      2     (1 + [ ]) * (3 + 4)
        3          [ ] * (3 + 4)
      3 + 4            3 * [ ]
    3                   3 * ([ ] + 4)
  4                        3 * (3 + [ ])
7                                     3 * [ ]
21                                      [ ]
```
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of \((1 + 2) \times (3 + 4)\)

\[
\begin{align*}
(1 + 2) \times (3 + 4) & \quad [ ] \\
1 + 2 & \quad [ ] \times (3 + 4) \\
1 & \quad ([ ] + 2) \times (3 + 4) \\
2 & \quad (1 + [ ]) \times (3 + 4) \\
3 & \quad [ ] \times (3 + 4) \\
3 + 4 & \quad 3 \times [ ] \\
3 & \quad 3 \times ([ ] + 4) \\
4 & \quad 3 \times (3 + [ ]) \\
7 & \quad 3 \times [ ] \\
21 & \quad [ ]
\end{align*}
\]
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of \((1 + 2) \times (3 + 4)\)

\[
\begin{align*}
(1 + 2) \times (3 + 4) & \quad [ 

1 + 2 & \quad [ ] \times (3 + 4) \\
1 & \quad ( [ ] + 2) \times (3 + 4) \\
2 & \quad (1 + [ ]) \times (3 + 4) \\
3 & \quad [ ] \times (3 + 4) \\
3 + 4 & \quad 3 \times [ ] \\
3 & \quad 3 \times ([ ] + 4) \\
4 & \quad 3 \times (3 + [ ]) \\
7 & \quad 3 \times [ ] \\
21 & \quad [ ]
\end{align*}
\]
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of \((1 + 2) \times (3 + 4)\)

\[
\begin{align*}
(1 + 2) \times (3 + 4) & \quad \text{[ ]} \\
1 + 2 & \quad \text{[ ]} \times (3 + 4) \\
1 & \quad ([ ] + 2) \times (3 + 4) \\
2 & \quad (1 + [ ]) \times (3 + 4) \\
3 & \quad [ ] \times (3 + 4) \\
3 + 4 & \quad 3 \times [ ] \\
3 & \quad 3 \times ([ ] + 4) \\
4 & \quad 3 \times (3 + [ ]) \\
7 & \quad \ldots \\
21 & \quad \ldots
\end{align*}
\]
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of $(1 + 2) \times (3 + 4)$

```
(1 + 2) \times (3 + 4)          [ ]
1 + 2                           [ ] \times (3 + 4)
1                              ([ ] + 2) \times (3 + 4)
2                             (1 + [ ]) \times (3 + 4)
3                               [ ] \times (3 + 4)
3 + 4                         3 \times [ ]
3                           3 \times ([ ] + 4)
4                          3 \times (3 + [ ])
7                          3 \times [ ]
21                         [ ]
```
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- **Example**: left-to-right evaluation of \((1 + 2) \times (3 + 4)\)

```
(1 + 2) \times (3 + 4)  \quad [ ]
1 + 2                   [ ] \times (3 + 4)
1                      ([ ] + 2) \times (3 + 4)
2                      (1 + [ ]) \times (3 + 4)
3                      [ ] \times (3 + 4)
3 + 4                   3 \times [ ]
3                      3 \times ([ ] + 4)
4                      3 \times (3 + [ ])
7
21
```
Continuation: the “rest of the computation” at a given point of a program execution

Example: left-to-right evaluation of \((1 + 2) \ast (3 + 4)\)

\[
\begin{align*}
(1 + 2) \ast (3 + 4) & \quad [ ] \\
1 + 2 & \quad [ ] \ast (3 + 4) \\
1 & \quad ([ ] + 2) \ast (3 + 4) \\
2 & \quad (1 + [ ]) \ast (3 + 4) \\
3 & \quad [ ] \ast (3 + 4) \\
3 + 4 & \quad 3 \ast [ ] \\
3 & \quad 3 \ast ([ ] + 4) \\
4 & \quad 3 \ast (3 + [ ]) \\
7 & \quad 3 \ast [ ] \\
21 & \quad [ ]
\end{align*}
\]
Continuations

- **Continuation**: the “rest of the computation” at a given point of a program execution

- Example: left-to-right evaluation of $(1 + 2) \times (3 + 4)$

\[
\begin{align*}
(1 + 2) \times (3 + 4) & \quad [ ] \\
1 + 2 & \quad [ ] \times (3 + 4) \\
1 & \quad ([ ] + 2) \times (3 + 4) \\
2 & \quad (1 + [ ]) \times (3 + 4) \\
3 & \quad [ ] \times (3 + 4) \\
3 + 4 & \quad 3 \times [ ] \\
3 & \quad 3 \times ([ ] + 4) \\
4 & \quad 3 \times (3 + [ ]) \\
7 & \quad 3 \times [ ] \\
21 & \quad [ ]
\end{align*}
\]
Continuations in Functional Languages

- Explicit: continuation-passing style (CPS)
  - Morris, Fischer, Reynolds, Plotkin, Steele

- Implicit: access to control flow through control operators
  - Landin’s J, Reynolds’s escape, Sussman & Steele’s catch, call/cc in Scheme and SML/NJ, Felleisen’s C
Functions accept an additional functional parameter: the continuation

Continuations encode the remaining computation

No nested function applications (all calls are tail calls)

Intermediate results are named

Computations are sequentialized
Continuation-Passing Style: Example

Computing list product in CPS

```ml
fun product_cps (xs, top)
  = let fun walk (nil, k)
        = k 1
        | walk (0 :: xs, k)
        = top 0
        | walk (x :: xs, k)
        = walk (xs, fn v => k (x * v))
  in walk (xs, top) end
```
Control Operators

- Power of CPS in direct style (DS)
- Access to the current continuation as a first-class value
- Jumps in DS: `callcc (K)`
Control Operators: Example

- Computing list product in DS

```plaintext
fun product_ds xs = callcc
  (fn top => let fun walk nil = 1
   | walk (0 :: xs) = top 0
   | walk (x :: xs) = x * (walk xs)
   in walk xs end)
```

- Backtracking, coroutines, multiprocessing, etc.
Semantics of Control Operators

- Higher-order evaluator in CPS (continuation semantics) and CPS transformation

\[
\overline{x} = \lambda k. k x
\]

\[
\overline{\lambda x. e} = \lambda k. k (\lambda x k \overline{e} k)
\]

\[
\overline{e_0 e_1} = \lambda k \overline{e_0} (\lambda v_0. \overline{e_1} (\lambda v_1. v_0 v_1 k))
\]

\[
\overline{K x. e} = \lambda k \overline{e} \{(\lambda v k'. k v)/x\} k
\]

CPS transformation

DS program with control operators

program in CPS

DS transformation
Outline

1 Introduction
   ■ Functional Programming and Continuations
   ■ Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   ■ Static Delimited Continuations
   ■ Dynamic Delimited Continuations
   ■ Comparison and Issues

3 Dynamic Continuation-Passing Style
   ■ New Operational Characterization
   ■ Higher-Order Characterization
   ■ Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Continuations vs. Delimited Continuations

- Continuations (undelimited, abortive)
  - Represent the entire “rest of the computation”
  - Model tail calls (i.e., jumps)

- Delimited continuations (composable, functional)
  - Represent a prefix of the “rest of the computation”
  - Model non-tail calls (i.e., composition of continuations)
Delimited-Control Operators

- **Control delimiter**: delimits the current continuation

- **Delimited-control operator**: gives access to a delimited continuation as a first-class value

- **Proposed control operators**
  - Felleisen ‘88
  - Danvy & Filinski ‘90
  - Queinnec & Serpette ‘91
  - Hieb et al. ’93
  - Sitaram ‘93
  - Moreau & Queinnec ‘94
  - Gunter et al. ’95
  - Dybvig et al. ‘05
Continuations vs. Delimited Continuations, ctd.

- **call-with-current-continuation (callcc)**

  \[1 + 10 + \text{callcc } k \ (k \ 0 + k \ 100)\]

  \[
  \Rightarrow \ \text{let } k = \text{fn } v \Rightarrow \text{abort } (11 + v) \ \text{in} \ k \ 0 + k \ 100 \ \text{end}
  \]

  \[
  \Rightarrow 11
  \]

- **call-with-current-delimited-continuation (callcdc) and delimit-continuation (dc)**

  \[\text{dc } (1 + \text{dc } (10 + \text{callcdc } k \ (k \ 0 + k \ 100)))\]

  \[
  \Rightarrow \ \text{let } k = \text{fn } v \Rightarrow \text{dc } (10 + v)
  \ \text{in} \ \text{dc } (1 + \text{dc } (k \ 0 + k \ 100)) \ \text{end}
  \]

  \[
  \Rightarrow \ \text{dc } (1 + \text{dc } (\text{dc } 10 + \text{dc } 110))
  \]

  \[
  \Rightarrow 121
  \]
Motivation for Delimited Continuations

- They have numerous applications
  - Non-deterministic programming (Kiselyov et al. ’05)
  - Computational monads (Filinski ’94, ’96, ’99)
  - Partial evaluation (Danvy ’96, ’98; Balat & Danvy ’98)
  - Normalization by evaluation (Dybjer & Filinski ’00)
  - Code generation (Thiemann ’99)
  - Concurrency (Hieb and Dybvig ’90, ’93)
  - Mobile computing (Sumii ’00)
  - Linguistics (Barker ’04, Shan ’04)
  - Web interactions (Queinnec ’00)

- They “refine” undelimited continuations
  - More expressive
  - Can simulate arbitrary computational effects (Filinski ’94)
  - Full abstraction w.r.t. CPS models (Sitaram & Felleisen ’90)
Contrasting semantics and applications
  - Static delimited continuations (Danvy & Filinski)
  - Dynamic delimited continuations (Felleisen)

A new CPS-based characterization of dynamic continuations
Outline

1. Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2. Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3. Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4. Summary, Related Work and Possible Extensions
Outline

1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Explicit

- Continuation-composing style (CCS)
- Continuation-passing style with 2 layers of continuations (2CPS)

Implicit

- Static delimited-control operators
Functions accept an additional functional parameter: the delimited continuation

Delimited continuation encodes a prefix of the remaining computation

Nested applications model composition of continuations
Listing list prefixes in CCS

```ml
fun prefixes_ccs xs
    = let fun walk (nil, k)
        = nil
        | walk (x :: xs, k)
        = (k (x :: nil))
        ::
        (walk (xs, fn v => k (x :: v)))
    in walk (xs, fn v => v)
    end
```

CPS-transforming this function yields a function in 2CPS
Listing list prefixes in 2CPS

fun prefixes_2cps l =
  let
    fun walk (nil, k, mk)
      = mk nil
    | walk (x :: xs, k, mk)
      = k (x :: nil,
          fn vs =>
            walk (xs,
                fn (vs', mk') => k (x :: vs', mk'),
                fn vss => mk (vs :: vss)))
  in walk (l, fn (vs, mk) => mk vs, fn vss => vss) end
Control operator \texttt{shift (S)} and control delimiter \texttt{reset (⟨·⟩)} (Danvy & Filinski)

Motivation

- Power of continuation-composing style in direct style
- Success and failure continuations model of backtracking in direct style

\begin{verbatim}
fun fail ()
  = shift (fn k => () )

fun amb (c1, c2)
  = shift (fn k => reset (fn () => k (c1 ())) ;
      reset (fn () => k (c2 ()) ) )
\end{verbatim}
Listing list prefixes in DS

fun prefixes_ds xs = let
    fun walk nil = shift (fn k => nil)
    | walk (x :: xs) = shift (fn k =>
        (reset
            (fn () => k (x :: nil)))
        ::
        (reset
            (fn () => k (x :: (walk xs)))))
    in reset (fn () => walk xs) end
Semantics of shift and reset

- Higher-order evaluator and CPS transformation

\[
\langle e \rangle = \lambda k. k (\bar{e} (\lambda v. v))
\]

\[
\bar{S}x.e = \lambda k. \bar{e}((\lambda v. k' (k v))/x) (\lambda v. v)
\]

- Studies based on CPS
  - Type structure: Danvy & Filinski ’89, Murthy ’92, Wadler ’94, Ariola et al. ’04, Asai & Kameyama ’07
  - Reasoning about programs: Hasegawa & Kameyama ’03, Kameyama ’04
  - Partial evaluation: Asai ’02, ’04, ’05
  - Operational semantics: Danvy & Yang ’99, Biernacka et al. ’05
Functional correspondence (PPDP ’03)

Defunctionalizing a higher-order evaluator in CPS yields an abstract machine

Abstract machines in defunctionalized form can be refunctionalized into higher-order evaluators in CPS

Defunctionalization (Reynolds ’72)

Transforms higher-order programs into equivalent first-order ones by replacing anonymous function spaces with

- first-order data type
- dispatch function over this data type

Programs in defunctionalized form: defunctionalized values are consumed only by the dispatch function!
Defunctionalization: Example

- Factorial in CPS

```haskell
fun fact_cps n = 
  let fun fact (0, k) = k 1 
    | fact (n, k) 
      = fact (n - 1, fn v => k (n * v)) 
  in fact (n, fn v => v) end
```

- Factorial in CPS, defunctionalized

```haskell
datatype cont = CONT0 | CONT1 of int * cont

fun fact_cps_def n = 
  let fun fact (0, k) = continue (k, 1) 
    | fact (n, k) = fact (n - 1, CONT1 (n, k)) 
    and continue (CONT0, v) = v 
    | continue (CONT1 (n, k), v) 
      = continue (k, n * v) 
  in fact (n, CONT0) end
```
Abstract Machine for shift/reset

- Defunctionalized evaluator in 2CPS (LMCS ’05)
  - continuation - the current delimited continuation
  - meta-continuation - the rest

- Language
  \[ e ::= x \mid \lambda x.e \mid e_0 e_1 \mid \langle e \rangle \mid Sx.e \]

- Values
  \[ v ::= [x, e, \rho] \mid C_1 \]

- Environments
  \[ \rho ::= \rho_{mt} \mid \rho\{x \mapsto v\} \]

- Contexts
  \[ C_1 ::= END \mid ARG((e, \rho), C_1) \mid FUN(v, C_1) \]

- Meta-contexts
  \[ C_2 ::= \text{nil} \mid C_1 :: C_2 \]
Abstract Machine for shift/reset (2)

- Transitions \((eval, cont_1, cont_2)\):

\[
\begin{align*}
\langle x, \rho, C_1, C_2 \rangle_{eval} & \Rightarrow \langle C_1, \rho(x), C_2 \rangle_{cont_1} \\
\langle \lambda x.e, \rho, C_1, C_2 \rangle_{eval} & \Rightarrow \langle C_1, [x, e, \rho], C_2 \rangle_{cont_1} \\
\langle e_0 e_1, \rho, C_1, C_2 \rangle_{eval} & \Rightarrow \langle e_0, \rho, \text{ARG}((e_1, \rho), C_1), C_2 \rangle_{eval} \\
\langle \langle e \rangle, \rho, C_1, C_2 \rangle_{eval} & \Rightarrow \langle e, \rho, \text{END}, C_1 :: C_2 \rangle_{eval} \\
\langle Sx.e, \rho, C_1, C_2 \rangle_{eval} & \Rightarrow \langle e, \rho\{x \mapsto C_1\}, \text{END}, C_2 \rangle_{eval} \\
\langle \text{END}, v, C_2 \rangle_{cont_1} & \Rightarrow \langle C_2, v \rangle_{cont_2} \\
\langle \text{ARG}((e, \rho), C_1), v, C_2 \rangle_{cont_1} & \Rightarrow \langle e, \rho, \text{FUN}(v, C_1), C_2 \rangle_{eval} \\
\langle \text{FUN}([x, e, \rho], C_1), v, C_2 \rangle_{cont_1} & \Rightarrow \langle e, \rho\{x \mapsto v\}, C_1, C_2 \rangle_{eval} \\
\langle \text{FUN}(C'_1, C_1), v, C_2 \rangle_{cont_1} & \Rightarrow \langle C'_1, v, C_1 :: C_2 \rangle_{cont_1} \\
\langle C_1 :: C_2, v \rangle_{cont_2} & \Rightarrow \langle C_1, v, C_2 \rangle_{cont_1} \\
\langle \text{nil}, v \rangle_{cont_2} & \Rightarrow v
\end{align*}
\]

- Extent of control operations determined \textit{statically}
1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Dynamic Delimited Continuations

- Control operator \textit{control} ($\mathcal{F}$) and control delimiter prompt (#) (Felleisen)

- Motivation
  - Refined calculus for reasoning about control operators

- Semantics
  - Abstract machine
  - Reduction semantics
Definitional Abstract Machine for control/prompt

- **Language**
  
  \[ e ::= x | \lambda x. e | e_0 e_1 | \#e | F x.e \]

- **Values**
  
  \[ v ::= [x, e, \rho] | C_1 \]

- **Environments**
  
  \[ \rho ::= \rho_{mt} | \rho\{x \mapsto v}\]  

- **Contexts**
  
  \[ C_1 ::= \text{END} | \text{ARG}((e, \rho), C_1) | \text{FUN}(v, C_1) \]

- **Concatenation of contexts**
  
  \[
  \begin{align*}
  \text{END} \star C'_1 &= C' \\
  (\text{ARG}((e, \rho), C_1)) \star C'_1 &= \text{ARG}((e, \rho), C_1 \star C'_1) \\
  (\text{FUN}(v, C_1)) \star C'_1 &= \text{FUN}(v, C_1 \star C'_1)
  \end{align*}
  \]

- **Meta-contexts**
  
  \[ C_2 ::= \text{nil} | C_1 :: C_2 \]
Transitions $(eval, cont_1, cont_2)$

$$
\langle x, \rho, C_1, C_2 \rangle_{eval} \Rightarrow \langle C_1, \rho(x), C_2 \rangle_{cont_1}
$$

$$
\langle \lambda x.e, \rho, C_1, C_2 \rangle_{eval} \Rightarrow \langle C_1, [x, e, \rho], C_2 \rangle_{cont_1}
$$

$$
\langle e_0 e_1, \rho, C_1, C_2 \rangle_{eval} \Rightarrow \langle e_0, \rho, ARG (\langle e_1, \rho \rangle, C_1), C_2 \rangle_{eval}
$$

$$
\langle \#, \rho, C_1, C_2 \rangle_{eval} \Rightarrow \langle e, \rho, END, C_1 :: C_2 \rangle_{eval}
$$

$$
\langle \mathcal{F} x.e, \rho, C_1, C_2 \rangle_{eval} \Rightarrow \langle e, \rho\{x \mapsto C_1\}, END, C_2 \rangle_{eval}
$$

$$
\langle END, v, C_2 \rangle_{cont_1} \Rightarrow \langle C_2, v \rangle_{cont_2}
$$

$$
\langle ARG (\langle e, \rho \rangle, C_1), v, C_2 \rangle_{cont_1} \Rightarrow \langle e, \rho, FUN (v, C_1), C_2 \rangle_{eval}
$$

$$
\langle \text{FUN} ([x, e, \rho], C_1), v, C_2 \rangle_{cont_1} \Rightarrow \langle e, \rho\{x \mapsto v\}, C_1, C_2 \rangle_{eval}
$$

$$
\langle \text{FUN} (C'_1, C_1), v, C_2 \rangle_{cont_1} \Rightarrow \langle C'_1 \star C_1, v, C_2 \rangle_{cont_1}
$$

$$
\langle C_1 :: C_2, v \rangle_{cont_2} \Rightarrow \langle C_1, v, C_2 \rangle_{cont_1}
$$

$$
\langle \text{nil}, v \rangle_{cont_2} \Rightarrow v
$$

Extent of control operations determined dynamically
Outline

1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Comparison

- Static composition

\[ \langle \text{FUN} (C'_1, C_1), v, C_2 \rangle_{\text{cont}_1} \Rightarrow \langle C'_1, v, C_1 :: C_2 \rangle_{\text{cont}_1} \]

- Dynamic composition

\[ \langle \text{FUN} (C'_1, C_1), v, C_2 \rangle_{\text{cont}_1} \Rightarrow \langle C'_1 \star C_1, v, C_2 \rangle_{\text{cont}_1} \]

- Different operational behavior

- Regarded as a minor difference ... 
  - Dynamic continuations can simulate static ones (JFP ’06): replace \( k \) with \( \lambda x.\#(k \ x) \)

- ... but it has considerable consequences
fun foo xs = 
    let fun visit nil = nil 
        | visit (x :: xs) 
            = visit (shift (fn k => x :: (k xs)))
    in reset (fn () => visit xs) end

In foo xs all the captured continuations represent
fn v => visit v

foo copies its argument
fun bar xs = 
    let fun visit nil = nil
        | visit (x :: xs)
        = visit (control (fn k => x :: (k xs)))
    in prompt (fn () => visit xs) end

In bar xs the successive captured continuations represent

    fn v => visit v
    fn v => visit (x1 :: v)
    fn v => visit (x2 :: x1 :: v)
    ...
    fn v => visit (xn :: ... :: x1 :: v)

bar reverses its argument
Beyond CPS?

- Contexts are consumed by \( \text{cont}_1 \) and ★

- Definitional abstract machine is not in defunctionalized form

- Dynamic delimited continuations therefore appear to be incompatible with traditional CPS
No correspondence to the traditional CPS-based notion of continuation: lack of intuition

No verification or inspiration by CPS: only direct-style programming and testing

No meaning-preserving program transformations

No continuation or monadic semantics

No structure to study other aspects (e.g., types)

No meaningful examples exploiting ⋆: existing ones can be easily implemented with static continuations
Can the dynamism of control and prompt be useful in practice?

Perhaps to program computational patterns that are incompatible with traditional CPS—but are there any?
■ Can the dynamism of control and prompt be useful in practice?

■ Perhaps to program computational patterns that are incompatible with traditional CPS—but are there any?
Breadth-first traversal in direct style without a queue
(IPL ’05, SoCP ’06)

datatype tree = LEAF
  | NODE of tree * int * tree

fun visit (LEAF, a)
  = a
  | visit (NODE (t1, x, t2), a)
  = control
      (fn k => let val a’ = k (x :: a)
       val a’’ = visit (t1, a’)
       in visit (t2, a’’)
       end)

fun breadth_first t
  = prompt (fn () => visit (t, nil))
Conjecture: dynamic continuations give rise to a "control queue"

How do we really know what the programs do? (except for testing)

We need more semantics to understand dynamic delimited continuations better!

Our goal: treatment parallel to that of static continuations
Conjecture: dynamic continuations give rise to a "control queue"

How do we really know what the programs do? (except for testing)

We need more semantics to understand dynamic delimited continuations better!

Our goal: treatment parallel to that of static continuations
Challenge 2: Semantics

- Conjecture: dynamic continuations give rise to a "control queue"

- How do we really know what the programs do? (except for testing)

- We need more semantics to understand dynamic delimited continuations better!

- Our goal: treatment parallel to that of static continuations
Outline

1 Introduction
   ■ Functional Programming and Continuations
   ■ Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   ■ Static Delimited Continuations
   ■ Dynamic Delimited Continuations
   ■ Comparison and Issues

3 Dynamic Continuation-Passing Style
   ■ New Operational Characterization
   ■ Higher-Order Characterization
   ■ Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Outline

1. Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2. Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3. Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4. Summary, Related Work and Possible Extensions
■ Put the definitional abstract machine for control/prompt in defunctionalized form

■ Eliminate the concatenation of contexts by introducing a trail of contexts

■ Contexts, trails of contexts, and meta-contexts

\[
C_1 ::= \text{END} \mid \text{ARG}((e, \rho), C_1) \mid \text{FUN}(v, C_1)
\]
\[
T_1 ::= \text{nil} \mid C_1 :: T_1
\]
\[
C_2 ::= \text{nil} \mid (C_1, T_1) :: C_2
\]

■ Values

\[
v ::= [x, e, \rho] \mid [C_1, T_1]
\]
Put the definitional abstract machine for control/prompt in defunctionalized form

Eliminate the concatenation of contexts by introducing a trail of contexts

Contexts, trails of contexts, and meta-contexts

\[
C_1 ::= \text{END} \mid \text{ARG}((e, \rho), C_1) \mid \text{FUN}(v, C_1)
\]

\[
T_1 ::= \text{nil} \mid C_1 :: T_1
\]

\[
C_2 ::= \text{nil} \mid (C_1, T_1) :: C_2
\]

Values

\[
v ::= [x, e, \rho] \mid [C_1, T_1]
\]
Experiment: New Abstract Machine

- Put the definitional abstract machine for control/prompt in defunctionalized form
- Eliminate the concatenation of contexts by introducing a trail of contexts
- Contexts, trails of contexts, and meta-contexts

\[
\begin{align*}
C_1 & ::= \text{END} \mid \text{ARG}((e, \rho), C_1) \mid \text{FUN}(v, C_1) \\
T_1 & ::= \text{nil} \mid C_1 :: T_1 \\
C_2 & ::= \text{nil} \mid (C_1, T_1) :: C_2
\end{align*}
\]

- Values

\[
v ::= [x, e, \rho] \mid [C_1, T_1]
\]
Transitions

\[ \langle #e, \rho, C_1, T_1, C_2 \rangle_{eval} \Rightarrow \langle e, \rho, \text{END}, \text{nil}, (C_1, T_1) :: C_2 \rangle_{eval} \]
\[ \langle \mathcal{F} x.e, \rho, C_1, T_1, C_2 \rangle_{eval} \Rightarrow \langle e, \rho \{ x \mapsto [C_1, T_1] \}, \text{END}, \text{nil}, C_2 \rangle_{eval} \]

\[ \langle \text{FUN} ([C'_1, T'_1], C_1), v, T_1, C_2 \rangle_{cont_1} \Rightarrow \langle C'_1, v, T'_1 @ (C_1 :: T_1), C_2 \rangle_{cont_1} \]

Equivalent to the definitional machine (bisimulation argument)

In defunctionalized form
Outline

1 Introduction
   ■ Functional Programming and Continuations
   ■ Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   ■ Static Delimited Continuations
   ■ Dynamic Delimited Continuations
   ■ Comparison and Issues

3 Dynamic Continuation-Passing Style
   ■ New Operational Characterization
   ■ Higher-Order Characterization
   ■ Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
- Higher-order evaluator: refunctionalized abstract machine
- Continuation+state-passing style
- Dynamic CPS transformation

\[
\overline{\#e} = \lambda k. \lambda t. k (\overline{e \, \theta \text{nil}}) \, t
\]

\[
\overline{F \, x. \, e} = \lambda k. \lambda t. \overline{e} \{ \lambda v. \lambda k'. \lambda t'. k \, v (t \, @ \, (k' :: \, t')) \} / x \} \, \theta \text{nil}
\]

\[
\theta = \lambda v. \lambda t. \text{case } t \\
\quad \text{of } \text{nil } \Rightarrow v \\
\quad \mid k :: t' \Rightarrow k \, v \, t'
\]
Revisiting Breadth-First Traversal

- Transform the program using dynamic CPS
- Defunctionalize continuations
- Result: a queue-based program performing breadth-first traversal
  - Confirmation: dynamic continuations give rise to the notion of “control queue”
Breadth-First Traversal with Queue

- Traversing a tree

  ```
  fun visit (LEAF, a, qf, qr)
  = continue (qf, qr, a)
  |
  visit (NODE (t1, x, t2), a, qf, qr)
  = continue (qf, qr @ [(t1, t2)], x :: a)
  ```

- Processing a queue

  ```
  and continue (NONE, nil, a)
  = a
  |
  continue (NONE, (t1, t2) :: qr, a)
  = visit (t1, a, SOME t2, qr)
  |
  continue (SOME t, qr, a)
  = visit (t, a, NONE, qr)
  ```
1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Continuation monad with recursive answer type

\[ D_o = \langle D, \text{unit}, \text{bind} \rangle \]

\[ D\alpha = (\alpha \to \text{ans}) \to \text{ans} \]

\[ \text{ans} = \text{list}[o \to \text{ans}] \to o \]

\[ \text{unit} : \alpha \to D(\alpha) \]

\[ \text{unit}(v) = \lambda k.kv \]

\[ \text{bind} : D(\alpha) \to (\alpha \to D(\beta)) \to D(\beta) \]

\[ \text{bind } cf = \lambda k.c(\lambda v.f v k) \]
Control operations in the monad

\[
\text{prompt} : \ Do \to\ Do \\
\text{prompt}(c) = \lambda k. \lambda t. k (c \theta \text{nil}) t
\]

\[
\text{control} : ((\alpha \to \ Do) \to \ Do) \to \ Do \to \ Da \\
\text{control}(e) = \lambda k. \lambda t. \text{let } c = \lambda v. \lambda k'. \lambda t'. k v (t @ (k' :: t')) \\
in\ e\ c\ \theta\ \text{nil}
\]

Monadic evaluator (Wadler ‘94)

Factorization through monadic metalanguage (Danvy & Hatcliff ’94)
Static Simulation

- Direct-style representation of the monad using monadic reflection (Filinski ’94, ’96)

- A simulation of control/prompt in terms of shift/reset (à la Shan ’04)

\[
\begin{align*}
\#e &= \langle \theta e \rangle \text{nil} \\
\mathcal{F} \, x.\, e &= S \, k.\, \lambda t.\, \text{let } x = \lambda v.\, S \, k'.\, \lambda t'.\, k \, v (t \odot (k' :: t')) \quad \text{in } \langle \theta e \rangle \text{nil}
\end{align*}
\]

- Static and dynamic continuations are equally expressive!
Through the static simulation using Filinski’s implementation of shift/reset

Through a simulation of control/prompt in terms of callcc and two pieces of mutable state (à la Filinski ’94)
1 Introduction
   - Functional Programming and Continuations
   - Delimited Continuations

2 Static vs. Dynamic Delimited Continuations
   - Static Delimited Continuations
   - Dynamic Delimited Continuations
   - Comparison and Issues

3 Dynamic Continuation-Passing Style
   - New Operational Characterization
   - Higher-Order Characterization
   - Static Simulation of Dynamic Continuations

4 Summary, Related Work and Possible Extensions
Summary

- New semantic foundations of dynamic continuations based on CPS
- Important byproducts
- Enabling technology: defunctionalization

If you want to find out more


Related Work

- Continuation+state-passing style for dynamic continuations extending Wand et al.’s idea of an algebra of contexts (Shan ’04)

- Monadic framework for delimited continuations with multiple prompts extending Moreau and Queinnec’s idea of a marked meta-continuation (Dybvig et al. ’05)

- Simulation of dynamic continuations in terms of static ones by reifying control effects and their absence into values of a discrimination union type (Kiselyov ’05)
Possible Extensions

- Dynamic hierarchy ($\text{control}_n$ and $\text{prompt}_n$)
  - Layered computational effects
  - Repeated Dynamic CPS transformation

- Other dynamic control operators ($\text{shift0}$, $\text{control0}$)
  - Identify applications
  - Adjust dynamic CPS

$$\langle \mathcal{F}_0 k. e, \rho, C_1, T_1, (C'_1, T'_1) :: C_2 \rangle \Rightarrow \langle e, \rho\{k \mapsto [C_1, T_1]\}, C'_1, T'_1, C_2 \rangle$$

- Type and effect systems
  - Delimited continuations can have different answer types (Murthy)
  - Additionally, control operations can change answer types of delimited continuations (Danvy and Filinski, Asai and Kameyama)
Thank You