

Delimited-Continuations Blues

Olivier Danvy

BRICS, University of Aarhus, Denmark

About the title

- Continuations: a functional representation of the rest of the computation; applying one is like a jump (“leave now, and never come back”).
- Delimited continuations: a functional representation of part of the rest of the computation; applying one is like a call (“I’ll be back”).
- Blues: “I am now convinced that delimited continuations are a bad idea.” (Simon Peyton Jones, October 2, 2005)

Plan of the talk

- Show that delimited continuations actually are not without interest.
- Generalize.

How are delimited continuations specified?

- With a calculus (Felleisen):
Stop at the prompt of the toplevel loop.
- With an abstract machine (Duba et al.):
Compose continuations by concatenating their representations.
- Through CPS (Danvy and Filinski):
CPS-transform one more time.

Issue: these three views do not uniquely determine what delimited continuations are.

Static vs. dynamic delimited continuations

Say you apply a delimited continuation
and then you abstract control.

What is abstracted?

Dynamic view: everything.

Static view: up to the point of application.

- Commonly presented as a “minor” difference.
- The same, however, was said about the control stack in LISP.

Example: what does this program do?

```
fun traverse xs =
let fun visit nil =
      nil
    | visit (x :: xs) =
      visit
        (call_with_current_delimited_continuation
          (fn k => x :: (k xs)))
in delimit_control (fn () => visit xs)
end
```

Highlights of the program

- The computation takes place within a control delimiter.
- The function traverses its input list.
- The function constructs an output list.

No trick:

- The input and the output lists have the same length.
- The input and the output lists contain the same elements.

So what does this program do?

Example: what does this program do?

```
fun traverse xs =
let fun visit nil =
      nil
    | visit (x :: xs) =
      visit
        (call_with_current_delimited_continuation
          (fn k => x :: (k xs)))
in delimit_control (fn () => visit xs)
end
```

Static view: it copies!

The captured context is always

```
fn v => visit v
```

and so the program is really

```
fun traverse xs =  
  let fun visit nil =  
        nil  
      | visit (x :: xs) =  
        x :: (visit xs)  
  in delimit_control (fn () => visit xs)  
  end
```

Example: what does this program do?

```
fun traverse xs =  
  let fun visit nil =  
        nil  
      | visit (x :: xs) =  
        visit  
          (call_with_current_delimited_continuation  
            (fn k => x :: (k xs)))  
  in delimit_control (fn () => visit xs)  
  end
```

Dynamic view: it reverses!

For the input list (1 2 3), the captured context is successively

```
fn v => visit v
```

```
fn v => 1 :: (visit v)
```

```
fn v => 2 :: 1 :: (visit v)
```

```
fn v => 3 :: 2 :: 1 :: (visit v)
```

So: Not a minor difference after all

- Can yield different results.
- Useful?
- Reasonable?

A solution in search of a problem

Dynamic delimited continuations:

- What would be a good problem?
- Can we reason about the solution?

A problem: breadth-first traversal

- Intuition: data queue \rightarrow control queue. (cf. Andrzej:
new effect = restricted usage pattern of more general effect)
- Code: recursive descent + control / prompt.
- Key idea: return before recursively traversing the subtrees.

The code (1/2)

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

```
structure CP  
= Control_and_Prompt  
  (type intermediate_answer = int list)
```

```
val control = CP.control  
val prompt = CP.prompt
```

The code (2/2)

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

fun bf t
= prompt (fn () => let val () = visit t
                  in nil end)
```

It works! (theme and variations)

- Return before traversing: breadth-first.
- Return after traversing: depth-first.

Not only it works, but it also scales:

Okasaki's breadth-first numbering pearl (BRICS RS-05-13).

However...

But no reasoning principles

“On theories like this we cannot rely.

Proof we need. **Proof!**”

Yoda

Reasoning with delimited continuations

- The abstract machine for *static* delimited continuations is in defunctionalized form.
- The abstract machine for *dynamic* delimited continuations is not in defunctionalized form.

Eureka (joint work w/ Biernacki and Millikin)

- Introduce a “trail of contexts.”
- The resulting machine is in defunctionalized form:
 - a continuation-based evaluator follows and
 - so does a dynamic CPS transformation.

Back to the breadth-first example

- CPS-transform it,
- defunctionalize it, and
- simplify it.

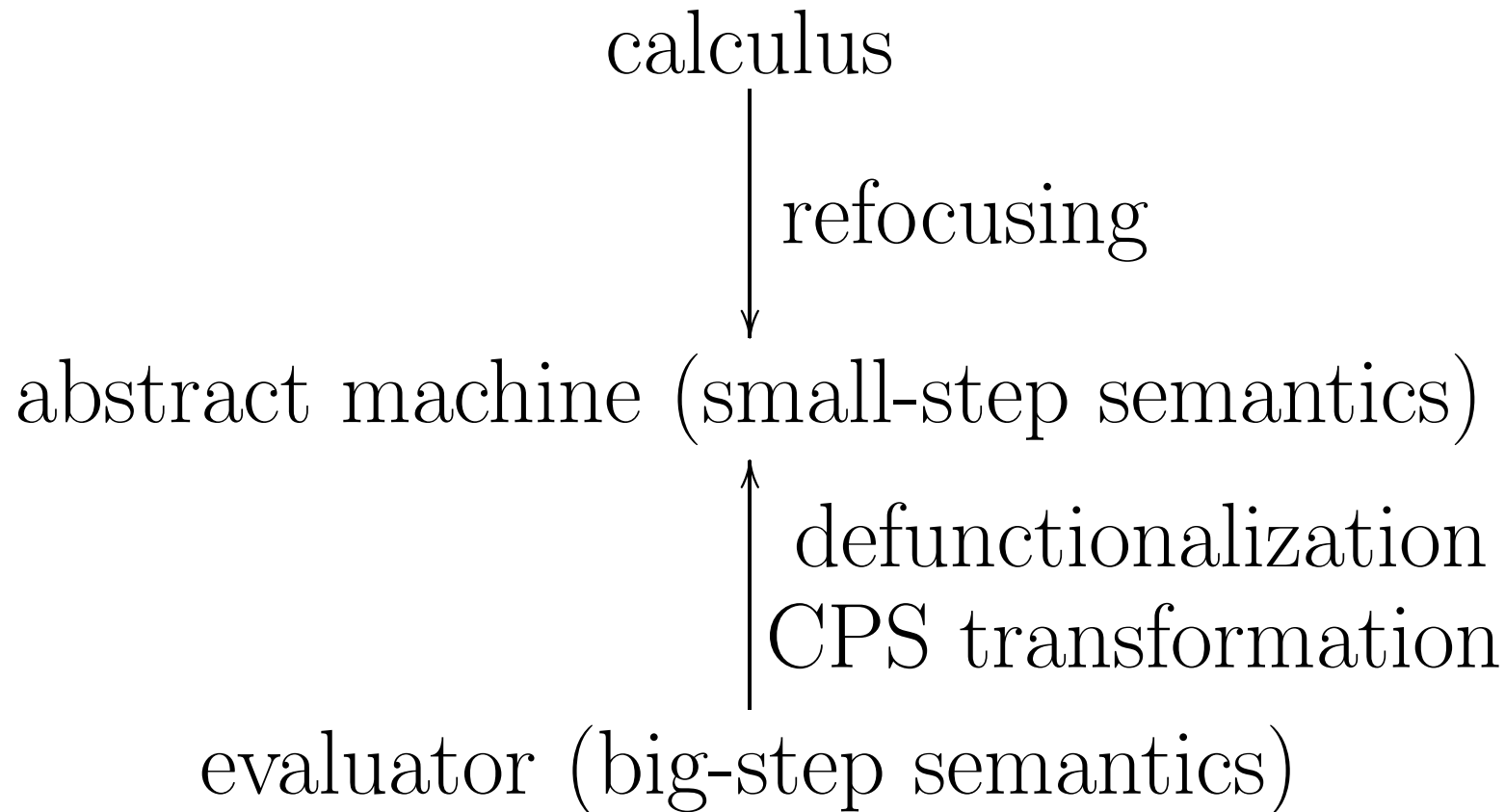
Result: an iterative queue-based program.

(And similarly for the depth-first example.)

And more (BRICS RS-05-16)

- A self-interpreter for the lambda-calculus with control and prompt.
- A monad for dynamic delimited continuations.
- A new simulation of control and prompt in terms of shift and reset (à la Ken Shan).

More generally



From this vantage point

- compare calculus and calculus
- compare abstract machine and abstract machine
- compare evaluator and evaluator

Kicking them blues

“I am now convinced that delimited continuations are a bad idea.”

- Dynamic continuations, I still don't know.
- But static continuations (and shift and reset) are in 1-1 correspondence with monads (cf. Filinski's PhD thesis).

So maybe not such a bad idea after all.

(cf. LISP and functional programming)

Looking at the method

- compare calculus and calculus
- compare abstract machine and abstract machine
- compare evaluator and evaluator

Environment machines (BRICS RS-05-15)

The syntactic and functional correspondences:

- Curien's calculus of closures \leftrightarrow environment machines
- normal-order calculus, the Krivine machine, CBN evaluator
- applicative-order calculus, the CEK machine, CBV evaluator
- normal order with generalized reduction, Krivine's machine
- applicative order with generalized reduction, ZINC

Computational effects (BRICS RS-05-22)

- normal order with generalized reduction + call/cc, Krivine's machine
- $\lambda\mu$ -calculus (de Groote)
- stack inspection (Fournier & Gordon, Clements & Felleisen)
- proper tail recursion (Clinger)
- call by need (Vuillemin)

a PhD-thesis generator
(actually: its antidote)