Refunctionalization at Work

Olivier Danvy

University of Aarhus, Denmark
(danvy@daimi.au.dk)

MPC'06 July 3, 2006

Defunctionalization: a change of representation

- Enumerate inhabitants of function space.
- Represent function space as a sum type and a dispatching apply function.
- Transform function declarations / applications into sum constructions / calls to apply.

Example: the factorial function in CPS

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


Example: the factorial function in CPS

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The continuation

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


All calls are tail calls

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

All sub-computations are trivial

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


The domain of answers

```
(*  fac : int * (int -> 'a) -> 'a  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The factorial program as a whole

```
(*  fac : int * (int -> int) -> int  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


Let us defunctionalize this factorial program.

The function space to defunctionalize

```
(*  fac : int * (int -> int) -> int  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


Inhabitants?

Who inhabits this function space?

The constructors

```
(*  fac : int * (int -> int) -> int  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```


The consumers

```
(*  fac : int * (int -> int) -> int  *)
fun fac (0, k)
    = k 1
  | fac (n, k)
    = fac (n - 1, fn v => k (n * v))

fun main n
    = fac (n, fn a => a)
```

The defunctionalized continuation

```
datatype cont = C0
              | C1 of cont * int

fun apply (C0, v)
    = v
  | apply (C1 (k, n), v)
    = apply (k, n * v)
```


Factorial in CPS, defunctionalized

```
fun fac (0, k)
    = apply (k, 1)
  | fac (n, k)
    = fac (n - 1, C1 (k, n))

fun main n
    = fac (n, C0)
```

Correctness

By structural induction on `n`,

using a logical relation over

the original continuation and

the defunctionalized continuation.

(Those who like this kind of things etc.)

Defunctionalization

- Introduced by John Reynolds in "Definitional Interpreters" (1972)

  `<www.brics.dk/~hosc/vol11/>`.

- Generalizes Peter Landin's notion of <u>closure</u> <u>conversion</u> (1964).

- Less used than closure conversion since.

Our thesis

- There is more to defunctionalization than an encoding, a “firstification.”
- Its left inverse, refunctionalization, is interesting.

Our thesis

- There is more to defunctionalization than an encoding, a “firstification.”
- Its left inverse, refunctionalization, is interesting.

Reference: Danvy and Nielsen,

“Defunctionalization at work” at PPDP 2001.

Our thesis

- There is more to defunctionalization than an encoding, a "firstification."
- Its left inverse, refunctionalization, is interesting.

Reference: Danvy and Nielsen,

"Defunctionalization at work" at PPDP 2001.

Latent question

How does one construct

programming or even semantic artifacts?

(e.g., an abstract machine)

Latent question

How does one construct
programming or even semantic artifacts?
(e.g., an abstract machine)

Our point: Defunctionalization

provides elements of answer.

The rest of this talk

- A series of examples illustrating defunctionalization and refunctionalization.
- A characterization of “defunctionalized form.”
- Hints for massaging a program into defunctionalized form.

Exercise: listing prefixes

Write a function

mapping a list to the list of its prefixes

whose last element satisfies a predicate.

Example, for the "always true" predicate:

$$[1, 2, 3] \longrightarrow [[1], [1, 2], [1, 2, 3]]$$

Example, for the "odd" predicate:

$$[1, 2, 3, 4, 5] \longrightarrow [[1], [1, 2, 3], [1, 2, 3, 4, 5]]$$

On listing prefixes

- finding the first prefix and
  finding all prefixes
- use a first-order accumulator and
  use a functional accumulator

```
find_first_prefix_a (p, xs) ≝
letrec visit (nil, a)
        = nil
       | visit (x :: xs, a)
        = let a′ = x :: a
          in if p x
             rev (a′, nil)
             else visit (xs, a′)
in visit (xs, nil)
```

27

```
find_all_prefixes_a (p, xs) ≝
letrec visit (nil, a)
        = nil
       | visit (x :: xs, a)
        = let a′ = x :: a
          in if p x
             (rev (a′, nil)) :: (visit (xs, a′))
             else visit (xs, a′)
in visit (xs, nil)
```

A functional accumulator

$$\begin{aligned} \mathit{hnil} &= \lambda xs.xs \\ \mathit{hcons} &= \lambda x.\lambda xs.x :: xs \end{aligned}$$

A novel representation of lists

and its application to the function "reverse"

John Hughes, IPL 22(3):141-144, 1986

$$\begin{array}{l}
\mathit{find_first_prefix_c1}\ (p,\ xs) \stackrel{\text{def}}{=} \\
\mathit{letrec}\ \mathit{visit}\ (\mathit{nil},\ k) \\
\qquad = \mathit{nil} \\
\qquad \mid \mathit{visit}\ (x :: xs,\ k) \\
\qquad = \mathit{let}\ k' = k \circ (\mathit{hcons}\ x) \\
\qquad\quad\ \mathit{in}\ \mathit{if}\ p\ x \\
\qquad\qquad\quad k'\ \mathit{nil} \\
\qquad\qquad\quad \mathit{else}\ \mathit{visit}\ (xs,\ k') \\
\mathit{in}\ \mathit{visit}\ \mathit{hnil}
\end{array}$$

$$
\begin{aligned}
&\mathit{find\_all\_prefixes\_c1}\,(p,\ xs) \stackrel{\text{def}}{=}\\
&\mathit{letrec}\ \mathit{visit}\,(\mathit{nil},\ k)\\
&\qquad\quad = \mathit{nil}\\
&\qquad\quad |\ \mathit{visit}\,(x :: xs,\ k)\\
&\qquad\quad = \mathit{let}\ k' = k \circ (\mathit{hcons}\ x)\\
&\qquad\qquad \mathit{in}\ \mathit{if}\ p\ x\\
&\qquad\qquad\quad (k'\ \mathit{nil}) :: (\mathit{visit}\,(xs,\ k'))\\
&\qquad\qquad\quad \mathit{else}\ \mathit{visit}\,(xs,\ k')\\
&\mathit{in}\ \mathit{visit}\ \mathit{hnil}
\end{aligned}
$$

How related are the two solutions?

Answer #1: they are just different.

How related are the two solutions?

Answer #2: one is the defunctionalized version

of the other.

Data type: list; apply function: reverse.

Almost in CPS

The functional accumulator

is a delimited continuation.

Almost in CPS

The functional accumulator

is a delimited continuation.

...shift and reset.

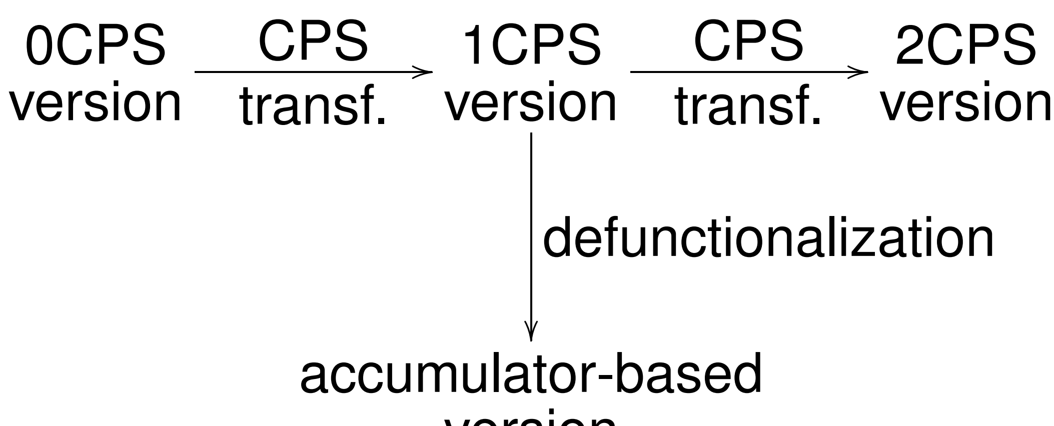
$$\mathit{find\_first\_prefix\_c0}\,(p,\ xs) \stackrel{\text{def}}{=}$$

$$\begin{array}{l}
\mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
\qquad\quad = \mathcal{S}\,k.\mathit{nil} \\
\qquad\ \mid \mathit{visit}\,(x :: xs) \\
\qquad\quad = x :: (\mathit{if}\ p\,x\ \mathit{then}\ \mathit{nil}\ \mathit{else}\ \mathit{visit}\,xs) \\
\mathit{in}\ \langle \mathit{visit}\,xs \rangle
\end{array}$$

$$\mathit{find\_all\_prefixes\_c0}\,(p,\ xs) \stackrel{\text{def}}{=}$$

$$
\begin{array}{l}
\mathit{letrec}\ \mathit{visit}\ \mathit{nil} \\
\qquad\quad = \mathcal{S}\,k.\mathit{nil} \\
\qquad\ \ \mid \mathit{visit}\,(x :: xs) \\
\qquad\quad = x :: \mathit{if}\ p\,x \\
\qquad\qquad\qquad \mathcal{S}\,k'.\langle k'\,\mathit{nil}\rangle :: \langle k'\,(\mathit{visit}\,xs)\rangle \\
\qquad\qquad\qquad \mathit{else}\ \mathit{visit}\,xs \\
\mathit{in}\ \langle \mathit{visit}\,xs\rangle
\end{array}
$$

Connections



CPS transformation

- Names intermediate results.
- Sequentializes their computation.
- Introduces first-class functions (continuations).

A simple example (1/3)

```
f x (g x)
```

A simple example (2/3)

```
f x (g x)

let v1 = f x
    v2 = g x
    v3 = v1 v2
in v3
```


A simple example (3/3)

```
f x (g x)

let v1 = f x          \k.f x (\v1.
    v2 = g x             g x (\v2.
    v3 = v1 v2           v1 v2 (\v3.
in v3                    k v3)))
```

The Fibonacci function (1/3)

```
fib n
= if n <= 1
  then n
  else fib(n - 1) + fib(n - 2)
```


The Fibonacci function (2/3)

```
fib n
= if n <= 1
  then n
  else let v1 = fib(n - 1)
           v2 = fib(n - 2)
       in v1 + v2
```

The Fibonacci function (3/3)

```
fib (n, k)
= if n <= 1
  then k n
  else fib(n - 1, \v1.
       fib(n - 2, \v2.
       k (v1 + v2)))
```


The Fibonacci function (4/3)

```
fib n = let v0 = n <= 1
in if v then n
   else let n1 = n - 1
            v1 = fib n1
            n2 = n - 2
            v2 = fib n2
        in v1 + v2
```

To CPS or not to CPS?

Q. When should we leave a function
in direct style?

To CPS or not to CPS?

Q. When should we leave a function
in direct style?

A. When it is pure and total.

To a man with a hammer...

Given $[x_1, ..., x_n]$ and $[y_1, ..., y_n]$,

compute $[(x_1, y_n), ..., (x_n, y_1)]$.

n is unknown.


```
fun cnv1 (xs,ys) =
let fun walk (nil,a)
          = continue (a,ys,nil)
        | walk (x::xs,a)
          = walk (xs,x::a)
    and continue (nil,nil,r)
          = r
        | continue (x::a,y::ys,r)
          = continue (a,ys,(x,y)::r)
in walk (xs,nil) end
```

```
fun cnv1 (xs,ys) =
let fun walk (nil,a)
        = continue (a,ys,nil)
      | walk (x::xs,a)
        = walk (xs,x::a)
    and continue (nil,nil,r)
        = r
      | continue (x::a,y::ys,r)
        = continue (a,ys,(x,y)::r)
in walk (xs,nil) end
```

51

```
fun cnv1 (xs,ys) =
let fun walk (nil,a)
        = continue (a,ys,nil)
      | walk (x::xs,a)
        = walk (xs,x::a)
    and continue (nil,nil,r)
        = r
      | continue (x::a,y::ys,r)
        = continue (a,ys,(x,y)::r)
in walk (xs,nil) end
```

In defunctionalized form

- the list is the data type
- `continue` is apply


```
fun cnv2 (xs,ys) =
let fun walk (nil,k)
          = k (ys,nil)
        | walk (x::xs,k)
          = walk (xs,fn (y::ys,r)
                         => k (ys,(x,y)::r))
      in walk (xs,fn (nil,r) => r) end
```

...CPS

Direct style:

```
fun cnv3 (xs,ys) =
let fun walk nil
        = (ys,nil)
      | walk (x::xs)
        = let val (y::ys,r) = walk xs
          in (ys,(x,y)::r) end
    val (nil,r) = walk xs
in r end
```


There and back again

joint work with Mayer Goldberg

ICFP 2002

Fundamenta Informaticae 66(4):397-413, 2005

Next: The SECD machine

- Why: it is canonical.
- What: a quadruple
  (stack, environment, control, dump).
- How: transitions.

State-transition function

- Pre-abstract machine: a transition function from non-accepting state to accepting or non-accepting state + a “trampoline” function.
- Abstract machine: a tail-recursive transition function (the transition function has been inlined in the trampoline function).

The source language

```
type ide = string
datatype term = LIT of int
              | VAR of ide
              | LAM of ide * term
              | APP of term * term
type program = term  (* closed *)
```


The environment

```
signature ENV =
sig
  type 'a env
  val mt : 'a env
  val ext : ide * 'a * 'a env
            -> 'a env
  val lookup : ide * 'a env -> 'a
end
```

Expressible and denotable values

```
datatype value
         = INT of int
         | SUCC
         | CLOSURE of ide * term
                        * value env
```


Initial environment

```
val e_init = ext ("succ", SUCC, mt)
```

The four components

- `stack : value list`
- `environment : value env`
- `control : directive list`

```
datatype directive
         = TERM of term
         | APPLY
```

- `dump : (stack * environment * control) list`

Evaluation by iterated transition

```
run (v :: nil, e’, nil, nil)
= v

run (v :: nil, e’, nil,
     (s, e, c) :: d)
= run (v :: s, e, c, d)

run (s, e, (TERM (LIT n)) :: c, d)
= run ((INT n) :: s, e, c, d)
```

```
run (s,e,(TERM (VAR x)) :: c,d)
= run ((lookup (x,e)) :: s,e,c,d)

run (s,e,(TERM (LAM (x,t))) :: c,d)
= run ((CLOSURE (x,t,e)) :: s,e,c,d)

run (s,e,(TERM (APP (t0,t1))) :: c,d)
= run (s,e,
(TERM t1) :: (TERM t0) :: APPLY :: c,
       d)
```



```
run (SUCC :: (INT n) :: s, e,
     APPLY :: c, d)
= run ((INT (n+1)) :: s, e, c, d)

run ((CLOSURE (x, t, e')) :: v :: s,
     e, APPLY :: c, d)
= run (nil, ext (x, v, e'),
       (TERM t) :: nil,
       (s, e, c) :: d)
```

Initialization of the SECD machine

```
fun evaluate0 t
    = run (nil,
           e_init,
           (TERM t) :: nil,
           nil)
```


Theorem (Plotkin, 1975)

It works.

All in all

The SECD machine is a mouthful:

- Are all cases accounted for?
- Are there any redundant clauses?

Disentangling the SECD machine

```
run_c : S * E * C * D -> value
run_d : value * D -> value
run_t : term *
        S * E * C * D -> value
run_a : S * E * C * D -> value
```

Four run functions

- Each function has one induction variable.
- Correctness proven by fixed-point induction.

A quote

From Hardy's "A Mathematician's Apology."

“there is a very high degree of *unexpectedness*, combined with *economy* and *inevitability*. The arguments take so odd and surprising a form; the weapons used seem so childishly simple when compared with the far-reaching results; but there is no escape from the conclusion. There are no complications of detail—one line of attack is enough in each case;”

The disentangled SECD machine

```
run_c : S * E * C * D -> value
run_d : value * D -> value
run_t : term *
        S * E * C * D -> value
run_a : S * E * C * D -> value
```

And then a miracle happens

The disentangled definition is defunctionalized:

- the control and the dump are two data types;
- `run_c` and `run_d` are their apply function.

An higher-order counterpart of the SECD machine

```
run_t : term *
        S * E * C * D -> value
run_a : S * E * C * D -> value

    C = S * E * D -> value
    D = S -> value
```

Guess what?

The refunctionalized SECD machine

is in CPS.

Back to direct style

```
run_t : term *
        S * E * C -> S
run_a : S * E * C -> S

    C = S * E -> S
```

Guess what?

The DS'ed refunctionalized SECD machine

uses a control delimiter.

(The body of a lambda-abstraction

is evaluated with an empty control stack.)

Back to direct style again

```
run_t : term *
        S * E -> S * E
run_a : S * E -> S * E
```

...a big-step operational semantics.

Another funny thing

Why is the interpreter threading a data stack?

Making do without a stack

```
run_t :  term * E -> V * E
run_a : V * V * E -> V * E
```

...another big-step operational semantics.

Guess what?

The result is in closure-converted form

(i.e., in defunctionalized form).

Higher-order counterpart

```
datatype value = INT of int
               | SUCC
               | FUN of value -> value
```

Guess what?

The evaluator is compositional.

...the valuation function

of a denotational semantics.

Denotational content of the SECD machine

- Environment-based.
- Callee-save.
- With a control delimiter.

  (Actually, an unnecessary control delimiter.)

```
fun eval (LIT n, e)
    = (INT n, e)

  | eval (VAR x, e)
    = (lookup (x, e), e)
```



```
  | eval (APP (t0, t1), e)
    = let val (v1, e) = eval (t1, e)
          val (v0, e) = eval (t0, e)
      in apply (v0, v1, e)
      end
```

```
eval (LAM (x, t), e)
= (FUN (fn v => #1 (
        reset (fn () =>
               eval (t,
                     extend (x,
                             v,
                             e))))),
     e)
```



```
apply (SUCC, INT n, e)
= (INT (n+1), e)

apply (FUN f, v, e)
= (f v, e)
```

Assessment

- All it took was to disentangle the SECD transition function.
- The rest (refunctionalization, direct-style transformation, direct-style transformation with a control delimiter, data-stack elimination, and closure unconversion) was mechanical.

The essence of the SECD machine

Essential: environment-based and callee-save.

Inessential: the stack,
the control, and
the dump.

Remark

Hindsight is an exact science.

What about reversing the transformation?

We mechanically get back the SECD machine.

What about reversing the transformation?

We mechanically get back the SECD machine.

What about trying with variants?

- de Bruijn indices.
- Left-to-right evaluation.
- Proper tail recursion.
- Call by name (use thunks).
- Call by need (thread heap of update thunks).

- An SEC machine (no control delimiter).
- An SC machine (no environment).
- A EC machine (no stack).
- A C machine (no environment and no stack).

Assessment

evaluator

↑ closure conversion

data-stack introduction

CPS transformation

↓ defunctionalization

SECD machine

Scaling up

From evaluation function to abstract machine

A canonical evaluator (caller-save)

```
datatype term
= IND of int  (* de Bruijn index *)
| ABS of term
| APP of term * term

datatype expval
= FUN of denval -> expval
withtype denval = expval
```

```
fun eval (IND n, e)
    = List.nth (e, n)
  | eval (ABS t, e)
    = FUN (fn v => eval (t, v :: e))
  | eval (APP (t0, t1), e)
    = let val (FUN f) = eval (t0, e)
      in f (eval (t1, e))
      end
```


John Reynolds's warning (1972)

Beware of the evaluation order

of the meta-language:

- Call by name yields call by name.
- Call by value yields call by value.

```
fun eval (IND n, e)
    = List.nth (e, n)
  | eval (ABS t, e)
    = FUN (fn v => eval (t, v :: e))
  | eval (APP (t0, t1), e)
    = let val (FUN f) = eval (t0, e)
      in f (eval (t1, e))
      end
```


John Reynolds's warning (1972)

Beware of the evaluation order

of the meta-language:

- Call by name yields call by name.
- Call by value yields call by value.

So we use thunks to simulate call by name.

Experiment 1: CBN

canonical CBN evaluator for λ-terms

↓ closure conversion

CPS transformation

defunctionalization

abstract machine

Experiment 1: CBN

canonical CBN evaluator for λ-terms

↓ closure conversion

CPS transformation

defunctionalization

Krivine's abstract machine

Krivine's abstract machine

The abstract machine

of theoreticians.

(see, eg, Chris Hankin's textbook

"Lambda calculi, a guide for computer scientists",

or again Pierre-Louis Curien, Pierre Crégut, etc.)

Experiment 2: CBV

canonical CBV evaluator for λ-terms

↓ closure conversion

CPS transformation

defunctionalization

abstract machine

Experiment 2: CBV

canonical CBV evaluator for λ-terms

↓ closure conversion

CPS transformation

defunctionalization

Felleisen et al.'s CEK abstract machine

The CEK abstract machine

The simplest abstract machine

of programming-language people.

Significance of the result

Krivine's machine and the CEK machine:

- Probably the two best-known abstract machines for the λ-calculus.
- Developed and presented independently.
- Yet they are defunctionalized interpreters for higher-order programming languages.

Flashback

John Reynolds's warning
about evaluation-order independence.

Flashback

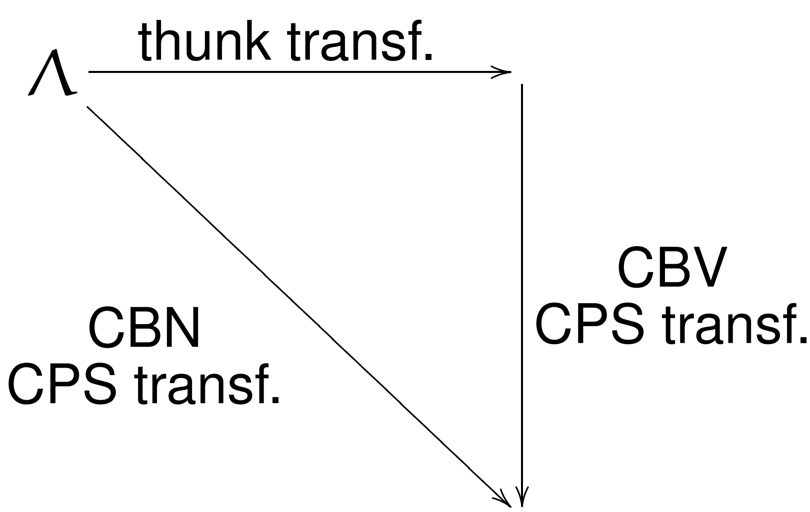
John Reynolds's warning

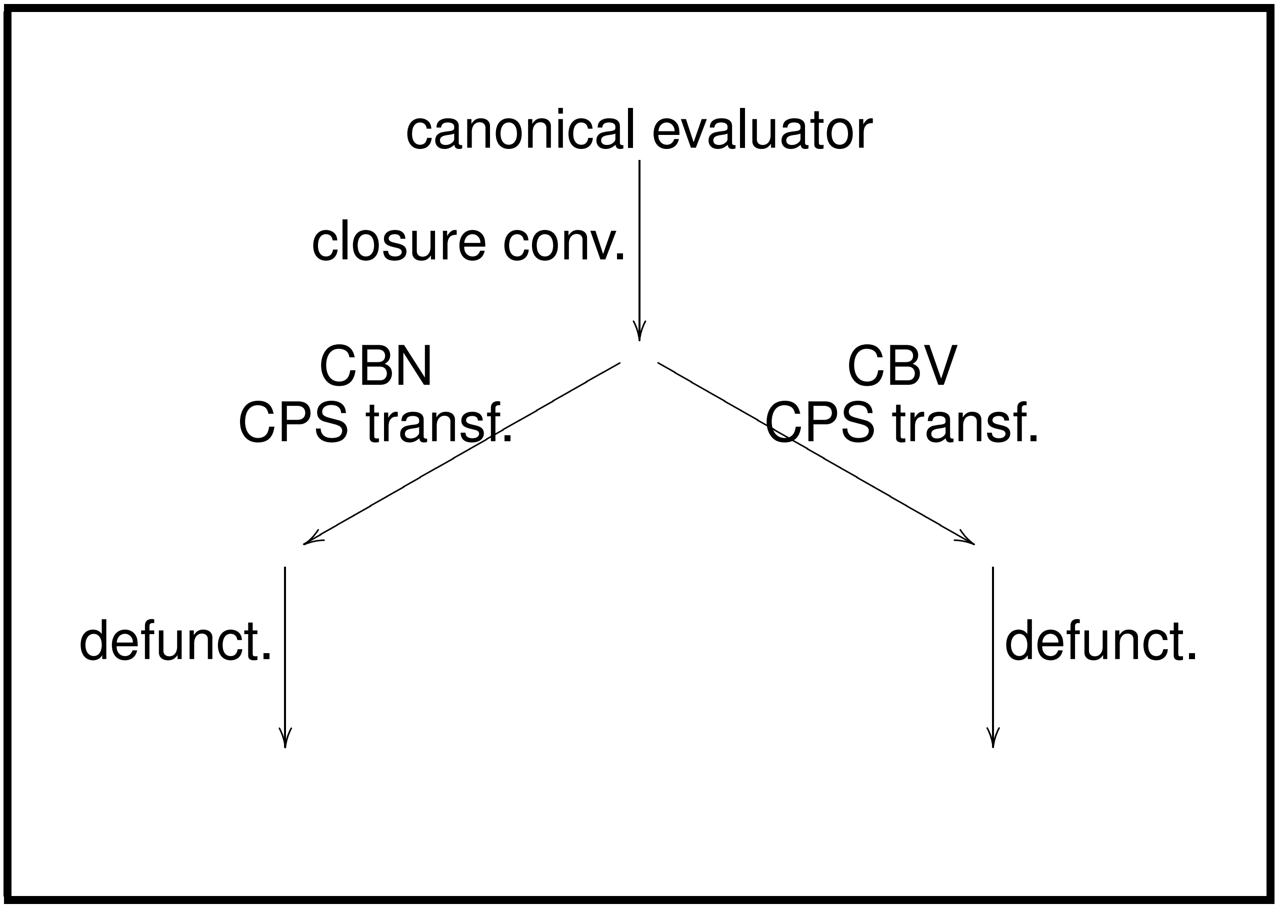
about evaluation-order independence.

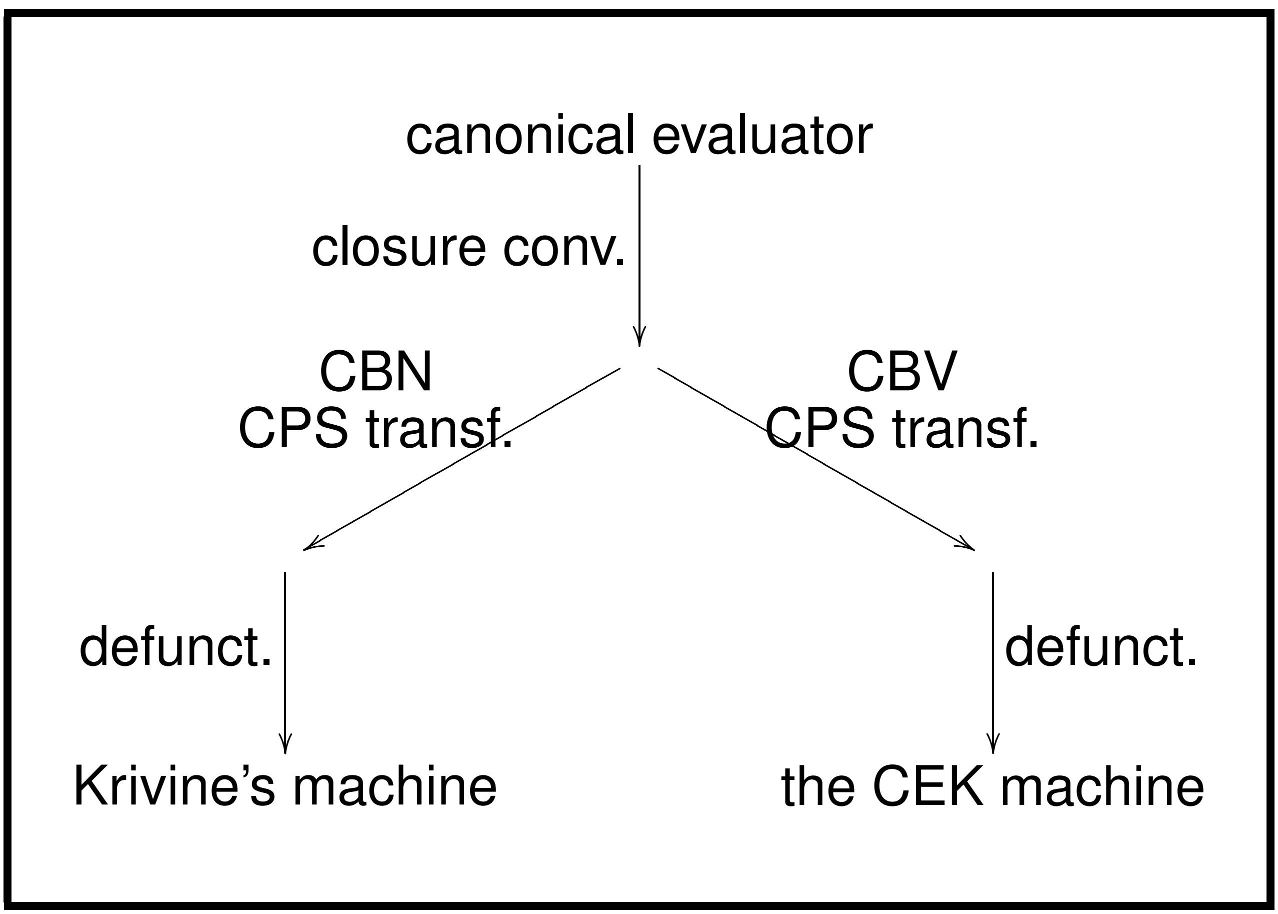
Let us use it constructively.

A factorization

(Hatcliff & Danvy, 1992–1997)




canonical evaluator
closure conv.
CBN
CPS transf.
CBV
CPS transf.
defunct.
defunct.


canonical evaluator
closure conv.
CBN
CPS transf.
CBV
CPS transf.
defunct.
defunct.
Krivine's machine
the CEK machine

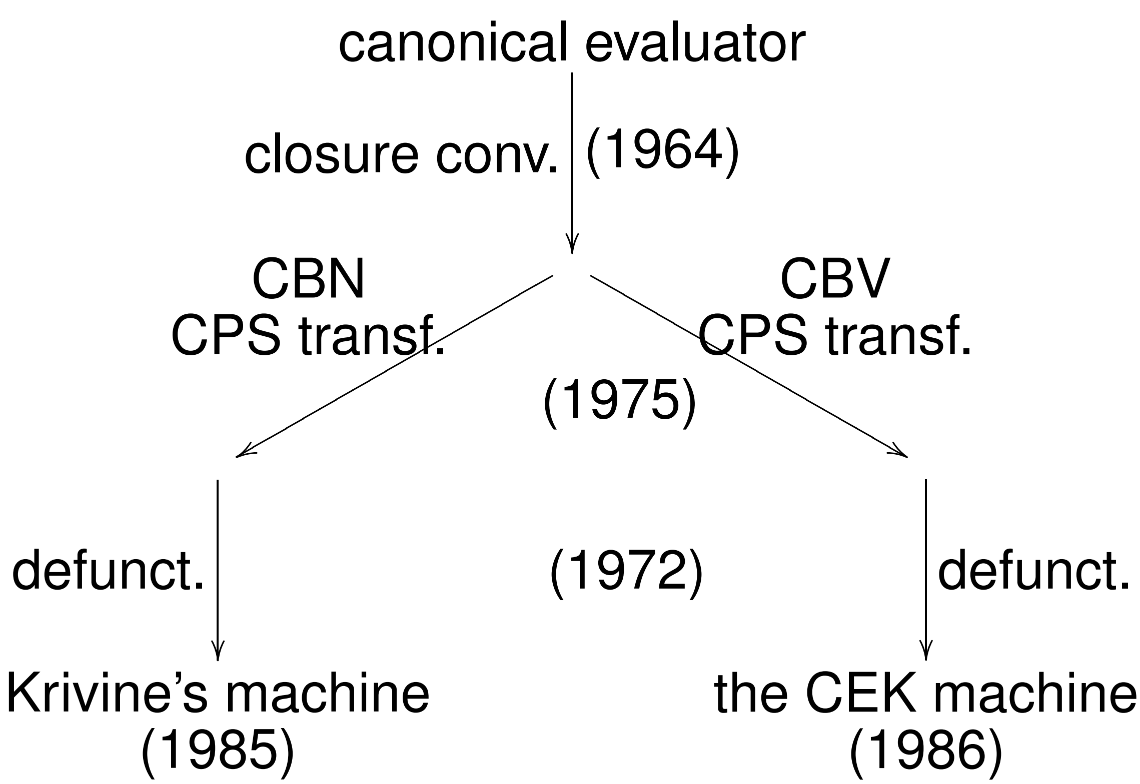
Consequence

Krivine's machine and the CEK machine

are not just discovered and invented.

They are two sides of the same coin,

which incidentally is the standard one.



Piet Hein's gentle reminder: T.T.T.

Put up in a place
where it's easy to see
the cryptic admonishment
T.T.T.
When you feel how depressingly
slowly you climb,
it's well to remember that
Things Take Time.

Models of abstract machines

- Eval-apply (CEK, etc.)
- Push-enter (KAM, etc.)

Models of abstract machines

- Eval-apply (CEK, etc.)
- Push-enter (KAM, etc.)

They appear naturally.

(inline the apply function in CBN)

Call by need
(built-in dynamic programming)

Call by need: Call by name +
heap of updatable thunks.

Result: A host of known implementation techniques and then some.

(see BRICS RS-03-20, IPL 90(5):223-232)

Computational effects

We build on Moggi's insight

as embodied in Wadler's interpreters.

One generic interpreter,

parameterized by a monad.

The style is in the monad.

The point

monadic evaluator + monad

↓ inlining (to make it 'styled')

closure conversion

CPS transformation

defunctionalization

abstract machine

Several detailed examples

Tech report BRICS RS-03-35:

- The identity monad.

  Result: the CEK machine.

- A lifted state monad.

  Result: the CEK machine
  with error and state.

Stack inspection

- A security mechanism to allow code with different levels of trust to interact in the same execution environment.
- Before execution, the source code is annotated with permissions.
- During execution, the call stack is inspected to check whether the required permissions are available.

Stack inspection

- See Section 6 in BRICS RS-03-35 (TCS 342(1):149-172, 2005)
- See Section 7 in BRICS RS-05-38 (to appear in TCS)

Yet

Not all abstract machines are in defunctionalized form. Examples:

- The SECD machine with the J operator.
- The CEK machine with dynamic delimited continuations.

Being in defunctionalized form

- several constructions sites
- one consumption site

Putting in defunctionalized form

No universal recipe. Handful of tricks:

- introducing auxiliary (first-order) functions
- delaying constructions
- glueing

The SECD machine with the J operator

- Landin's original version (1965)
  is incomplete.
- Burge's complete version (1975)
  is not in defunctionalized form.
- Felleisen's version (1987)
  is in defunctionalized form.

Felleisen's version

Refunctionalizing Felleisen's version
reveals a control delimiter (a "prompt").

See Danvy and Millikin, "A Rational Deconstruction of Landin's J Operator", IFL 2005 (extended version: BRICS RS-06-04).

Dynamic delimited continuations

See Biernacki, Danvy and Millikin, “A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations”, BRICS RS-05-16.

Conclusion

- Defunctionalization, like the lambda-calculus, has many applications.
- So does its left-inverse, refunctionalization.

Closing remarks

- Evaluation contexts are defunctionalized continuations.
-
-

Closing remarks

- Evaluation contexts are defunctionalized continuations.
- Reduction contexts are defunctionalized continuations.
-

Closing remarks

- Evaluation contexts are defunctionalized continuations.
- Reduction contexts are defunctionalized continuations.
- Most instances of the Zipper are defunctionalized continuations.

Closing remarks

- Evaluation contexts are defunctionalized continuations.
- Reduction contexts are defunctionalized continuations.
- Most instances of the Zipper are defunctionalized continuations.

Thank you.