

A Vademecum to Continuations

Olivier Danvy

BRICS, University of Aarhus, Denmark
(danvy@brics.dk)

 BRICS

EWSCS

February 28, 2005

Continuations

- What?
- What for?

Continuations represent
“the rest of the computation”

Example: denotational semantics

- Original goal: syntax-directed encoding in the λ -calculus.
- Denotational assumption: the encoding is compositional.
- Fine for expressions and sequence, but what about jumps?

Direct semantics of commands

$c \in \text{Com} ::= \text{skip} \mid c_1; c_2 \mid L : c \mid \text{goto } L \mid \dots$

$\mathcal{C} : \text{Com} \rightarrow \text{Env} \rightarrow \text{Sto} \rightarrow \text{Sto}$

$$\mathcal{C}[\text{skip}]\rho\sigma = \sigma$$

$$\mathcal{C}[c_1; c_2]\rho\sigma = \mathcal{C}[c_2]\rho(\mathcal{C}[c_1]\rho\sigma)$$

$$\mathcal{C}[\text{goto } L]\rho\sigma = ???$$

A catalyst (ca. 1970)

Strachey tells Wadsworth
about Mazurkiewicz's 'tail functions'.

Tilt!

- The denotation of a label should be such a tail function.
- Applying this tail function would continue after the label and yield the final answer.
- A more fitting name would thus be “continuation”.

Continuation semantics of commands

$$\mathcal{C} : \text{Com} \rightarrow \text{Env} \rightarrow \text{Sto} \rightarrow (\text{Sto} \rightarrow \text{Ans}) \rightarrow \text{Ans}$$

$$\mathcal{C}[\text{skip}] \rho \sigma \kappa = \kappa \sigma$$

$$\mathcal{C}[c_1; c_2] \rho \sigma \kappa = \mathcal{C}[c_1] \rho \sigma (\lambda \sigma'. \mathcal{C}[c_2] \rho \sigma' \kappa)$$

$$\mathcal{C}[\text{goto } L] \rho \sigma \kappa = \rho L \sigma$$

Other control operators

Now the denotation of other control operators is simple:

$$\mathcal{C}[\text{halt}] \rho \sigma \kappa = \sigma$$

if $\text{Ans} = \text{Sto}$.

All in all

Direct semantics: $\text{Sto} \rightarrow \text{Sto}$

A command is a store transformer.

Continuation semantics:

$$\left\{ \begin{array}{l} \text{Sto} \rightarrow (\text{Sto} \rightarrow \text{Ans}) \rightarrow \text{Ans} \\ (\text{Sto} \rightarrow \text{Ans}) \rightarrow \text{Sto} \rightarrow \text{Ans} \end{array} \right.$$

A command is a continuation transformer.

Continuation semantics for expressions

Given $t ::= b \mid t_1 \rightarrow t_2$,

we can embed types as follows:

$$\llbracket t \rrbracket = (\llbracket t \rrbracket' \rightarrow o) \rightarrow o$$

$$\llbracket b \rrbracket' = b$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket' = \llbracket t_1 \rrbracket' \rightarrow \llbracket t_2 \rrbracket \quad \text{call by value}$$

$$\llbracket t_1 \rightarrow t_2 \rrbracket' = \llbracket t_1 \rrbracket \rightarrow \llbracket t_2 \rrbracket \quad \text{call by name}$$

Connection with logic (1990)

Writing $\neg t$ instead of $t \rightarrow 0$,
Griffin and Murthy recognized
a double-negation translation.

To sum up: principles

- A continuation represents the rest of a computation.
- Useful for modelling control in denotational semantics.
- Connected to logic.

Back to the title

“Vade mecum” = “go with me” in Latin.

The many discoveries of continuations

John Reynolds

Lisp and Symbolic Computation 6(3/4), 1993.

`<http://www.brics.dk/~hosc>`

Applications to functional programming

Rather than writing functions of type $t_1 \rightarrow t_2$

one can write continuation-passing functions of type

$$t'_1 \rightarrow (t'_2 \rightarrow o) \rightarrow o$$

Result: continuation-passing style (CPS).

Why CPS?

- Format.
- Expressive power.

The format of CPS

- All subterms are trivial.
- All calls are tail calls.

A compiler writer is very happy about that
(Steele, Appel, Morrisett).

But is this format intrinsic to CPS?

- No: cf., e.g, Moggi's monadic normal forms (which were also multiply discovered).
- MLton.
- Monadic normal forms are in bijective correspondence with CPS.

Example: the factorial function

```
(* fac : int -> int *)  
fun fac 0  
    = 1  
  | fac n  
    = n * (fac (n - 1))  
  
fun main n  
    = fac n
```

CPS transformation

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

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)
```

A more substantial case

Consider a local function: $b_0 \rightarrow (b_1 + b_2)$

In CPS: $b_0 \rightarrow ((b_1 + b_2) \rightarrow o) \rightarrow o$

A more substantial case

Consider a local function: $b_0 \rightarrow (b_1 + b_2)$

In CPS: $b_0 \rightarrow ((b_1 + b_2) \rightarrow o) \rightarrow o$

Or again: $b_0 \rightarrow (b_1 \rightarrow o) \times (b_2 \rightarrow o) \rightarrow o$

Exercise: Calder mobiles

- Consider a binary tree of natural numbers.
- Compute whether it is well balanced.
- Do it in one traversal.

To sum up: practice

- CPS is useful in compilers.
- CPS is useful for functional programming.

In other words

Continuations are worth studying a bit more.

1. Definitional interpreters.
2. Expressive power (deletion vs. retention).
3. Formalization.

1: Definitional interpreters

Task: write a definitional self-interpreter.

Question: relation between defining evaluation order and defined evaluation order?

Example: $\text{eval} \llbracket \text{apply}(e_0, e_1) \rrbracket$
 $= \text{apply} (\text{eval} \llbracket e_0 \rrbracket, \text{eval} \llbracket e_1 \rrbracket)$

Reynolds's solution (1972)

Constrain the text of the definitional interpreter so that CBN and CBV cannot be distinguished.

Key point: actual parameters should be “trivial” (their evaluation should always converge).

The interpreter is in CPS.

2: More expressive power?

Procedure calls and returns work LIFO.

Call frames can be allocated:

- either in the heap (retention model),
- or on a stack (deletion model).

Is retention more expressive than deletion?

Example

Consider:

```
let val x = 10
in fn y => x + y end
```

What is the extent of `x`?

Fischer (1972)

A translation of any program from a heap-based implementation to a stack-based one.

Key point: define procedures that never return, but that call a continuation instead.

CPS again.

Vademecum for the CPS transformation

- The varieties of the CPS transformations:
there is at least one per evaluation order.
- Plotkin's three key theorems:
simulation, indifference, and translation.

Call-by-name CPS transformation

$$\llbracket x \rrbracket = \lambda k. x k \quad \text{– not just } x$$

$$\llbracket \lambda x. e \rrbracket = \lambda k. k(\lambda x. \llbracket e \rrbracket)$$

$$\llbracket e_0 e_1 \rrbracket = \lambda k. \llbracket e_0 \rrbracket (\lambda v_0. v_0 \llbracket e_1 \rrbracket k)$$

Call-by-value CPS transformation “with continuations last” (Plotkin)

$$\llbracket x \rrbracket = \lambda k. kx$$

$$\llbracket \lambda x. e \rrbracket = \lambda k. k(\lambda x. \llbracket e \rrbracket)$$

$$\llbracket e_0 e_1 \rrbracket = \lambda k. \llbracket e_0 \rrbracket (\lambda v_0. \llbracket e_1 \rrbracket (\lambda v_1. v_0 v_1 k))$$

Call-by-value CPS transformation “with continuations first” (Fischer)

$$\llbracket x \rrbracket = \lambda k. kx$$

$$\llbracket \lambda x. e \rrbracket = \lambda k. k(\lambda k. \lambda x. \llbracket e \rrbracket k)$$

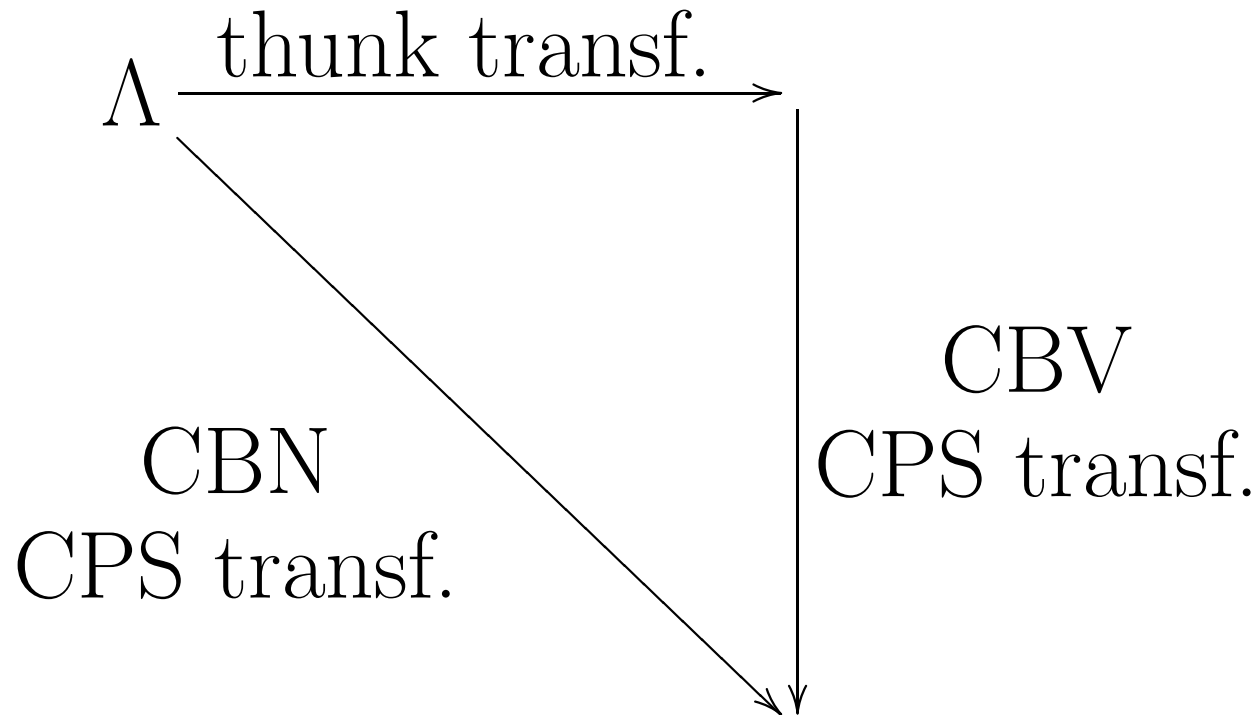
$$\llbracket e_0 e_1 \rrbracket = \lambda k. \llbracket e_0 \rrbracket (\lambda v_0. \llbracket e_1 \rrbracket (\lambda v_1. v_0 k v_1))$$

And many more

- call by value à la Algol 60 (Reynolds)
- left-to-right and right-to-left call by value
- mixed eval. orders (Danvy, Hatcliff, Nielsen)
- etc.

And their assorted double-negation translations
(Murthy).

A factorization (Hatcliff & Danvy, 1992-1997)



Plotkin (1975)

Simulation: evaluating a CPS'ed term gives the result of the original term, CPS'ed.

Indifference: a CPS term can be evaluated independently of the evaluation order.

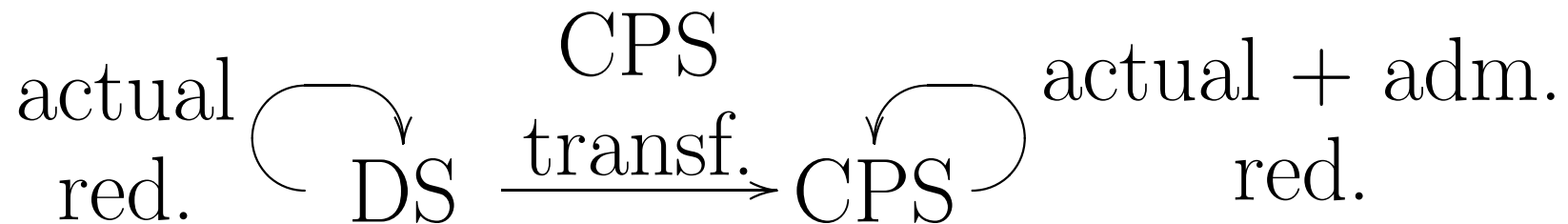
Translation: relation between equational theories of terms and of CPS'ed terms.

Actual and administrative reductions

Example:

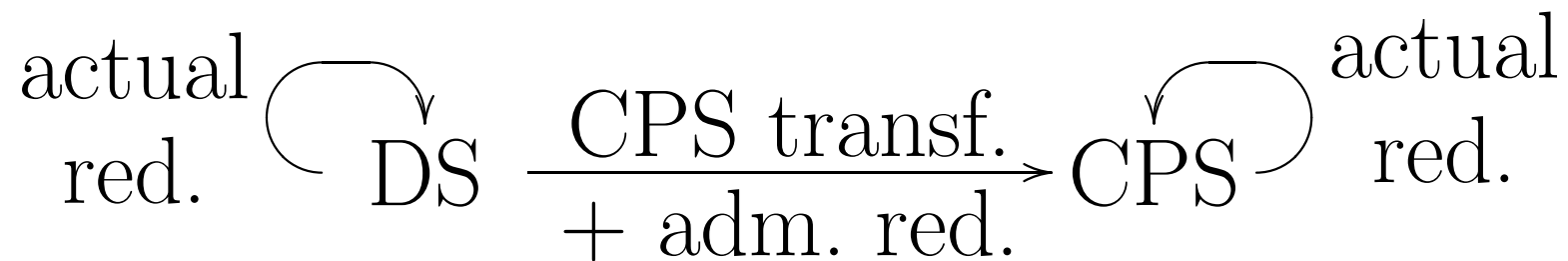
$[[\lambda x.x x]]$

$$= \lambda k.k \lambda x.\lambda k.(\lambda k.k x) \\ (\lambda v_0.(\lambda k.k x) (\lambda v_1.v_0 v_1 k))$$

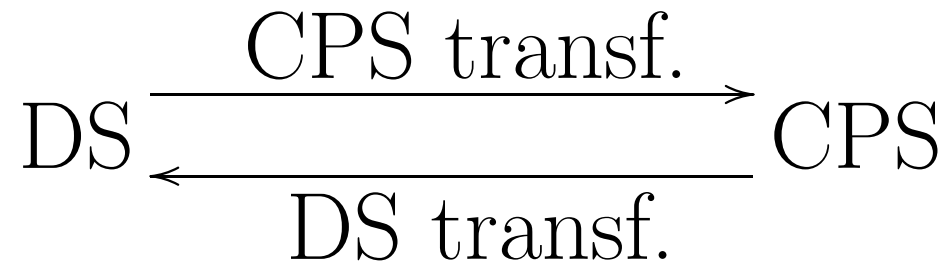


The “one-pass” approach

A CPS transformation performing
administrative reductions “at transformation time”
(Appel, Danvy and Filinski, Wand, ca. 1990)



The direct-style transformation



- Requires occurrence conditions over CPS programs (Danvy, Pfenning).
- Gave rise to INCLL (Pfenning, Polakow).

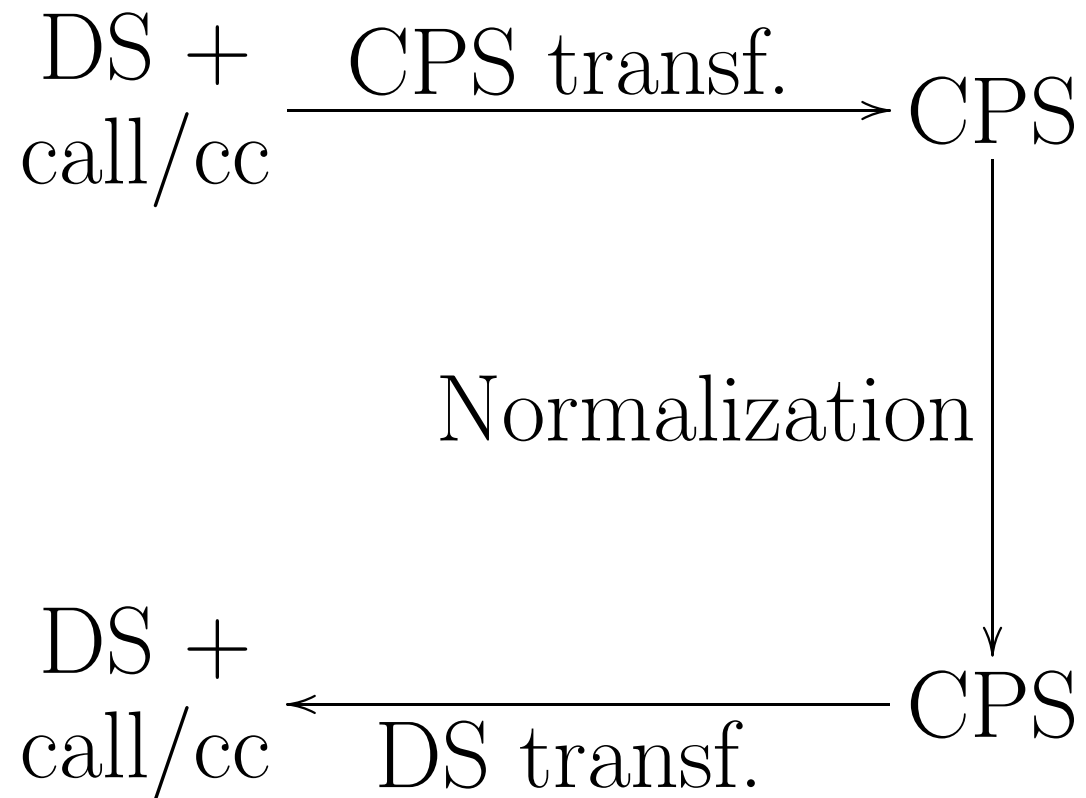
Control operators

Goal: to import the extra expressive power of continuations from CPS into direct style.

Examples: `goto`, `call/cc`.

N.B. Interesting typing consequences (Griffin).

Application to PE (Danvy and Lawall)



Delimited continuations

Idea: to define the rest of a sub-computation.

- prompts (Felleisen)
- iterated CPS transformation (Danvy and Filinski)
- etc.

Applications of delimited continuations

- Convenient for expressing backtracking.
- Instrumental for simulating monads (Filinski).
- Useful for type-directed partial evaluation.
- Logical contents currently explored by Kamenaya: there is more to delimited continuations than classical logic.

Other applications of continuations

- Reasoning about continuations (Felleisen, Thielecke).
- Program transformation (Wand).
- Compiler derivation (Clinger, Wand).
- Compiler generation (Appel, Oliva, Patterson, Wand).

More applications of continuations

- Program analysis (Nielson, Shivers, Consel and Danvy, Sabry and Felleisen, Damian and Danvy).
- Partial evaluation (Consel and Danvy, Bondorf, Lawall and Danvy).

Even more applications of continuations

- Multiprocessing (Wand).
- Coroutines (Haynes, Friedman).
- Operating-systems services (Mach 5.0).
- Parallelism (Le Métayer and Giorgi, Hieb and Dybvig, Moreau).

Implementing first-class continuations

A large body of work (Clinger et al., 1998).

Continuations as time passes by

1970's: Continuations storm in.

1980's: Continuations are explored, esp. in the Scheme community with call/cc.

1990's: A number of PhD theses are dedicated to continuations.

PhD theses

- Murthy (1991): CPS transformations are double-negation translations.
- Lawall (1994): CPS and DS transformations.
- Hatcliff (1994): the structure of CPS.
- Sabry (1994): equational models of CPS.

- Sitaram (1994): continuations and programming-language design.
- Moreau (1994): continuations and parallelism.
- Filinski (1996): continuations and monads.
- Thielecke (1997): categorical structure of CPS.
- Laird (1999): continuations and full abstraction.

- Führmann (2000): categorical models of control.
- Polakow (2001): INCLL.
- Nielsen (2001): defunctionalization.
- Berdine (2004): linear types.

...and probably more.

Milne and Mosses (70's)

“With continuation-passing and state-passing
we felt ready to conquer the world.”

Scheme community (80's)

“With call/cc and set!,
we can conquer the world.”

Moggi and Filinski (90's)

“Computational monads are the world.”

“Call/cc and set! can implement monads.”

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.

Defunctionalization example

```
(* 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 whole program

```
(* 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 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)
```

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.)

To a man with a hammer...

Given $[x_1, \dots, x_n]$ and $[y_1, \dots, y_n]$,

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

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 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

```

```
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 2005

To be continued

Thank you.