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

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

$$\mathcal{C} : \textsf{Com} \longrightarrow \textsf{Env} \longrightarrow \textsf{Sto} \longrightarrow \textsf{Sto}$$

$$\begin{aligned}
\mathcal{C}[\![\textsf{skip}]\!]\rho\sigma &= \sigma \\
\mathcal{C}[\![c_1; c_2]\!]\rho\sigma &= \mathcal{C}[\![c_2]\!]\rho(\mathcal{C}[\![c_1]\!]\rho\sigma) \\
\mathcal{C}[\![\textsf{goto}\ L]\!]\rho\sigma &= ???
\end{aligned}$$

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

$$\mathcal{C} : \textsf{Com} \longrightarrow \textsf{Env} \longrightarrow \textsf{Sto} \longrightarrow \big(\textsf{Sto} \longrightarrow \textsf{Ans}\big) \longrightarrow \textsf{Ans}$$

$$\begin{aligned}
\mathcal{C}[\![\textsf{skip}]\!]\,\rho\,\sigma\,\kappa &= \kappa\,\sigma \\
\mathcal{C}[\![c_1; c_2]\!]\,\rho\,\sigma\,\kappa &= \mathcal{C}[\![c_1]\!]\,\rho\,\sigma\,(\lambda\sigma'.\mathcal{C}[\![c_2]\!]\,\rho\,\sigma'\,\kappa) \\
\mathcal{C}[\![\textsf{goto}\ L]\!]\,\rho\,\sigma\,\kappa &= \rho\,L\,\sigma
\end{aligned}$$

Other control operators

Now the denotation of other control operators is simple:

$$\mathcal{C}[\![\mathsf{halt}]\!]\,\rho\,\sigma\,\kappa \;=\; \sigma$$

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

All in all

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

A command is a store transformer.

Continuation semantics:

$$\begin{cases} \text{Sto} \longrightarrow \big(\text{Sto} \longrightarrow \text{Ans}\big) \longrightarrow \text{Ans} \\ \big(\text{Sto} \longrightarrow \text{Ans}\big) \longrightarrow \text{Sto} \longrightarrow \text{Ans} \end{cases}$$

A command is a continuation transformer.

Continuation semantics for expressions

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

we can embed types as follows:

$$\llbracket t \rrbracket \; = \; (\llbracket t \rrbracket' \to o) \to o$$

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

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

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

Connection with logic (1990)

Writing $\neg t$ instead of $t \to o$,

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 $t_1 \longrightarrow t_2$

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: $b_0 \longrightarrow (b_1 + b_2)$

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

A more substantial case

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

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

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

Exercise: Calder mobiles

- Consider a binary tree of natural numbers.
- Compute whether it is well balanced.
- Do it $\boxed{\text{in one traversal}}$.

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 $\boxed{\text{defining}}$ evaluation order and $\boxed{\text{defined}}$ evaluation order?

Example: eval $[\![\text{apply}(e_0,\ e_1)]\!]$

$$= \text{apply (eval } [\![e_0]\!]\text{, eval } [\![e_1]\!]\text{)}$$

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

$$
\begin{aligned}
[\![x]\!] &= \lambda k.xk \qquad \text{– not just } x \\
[\![\lambda x.e]\!] &= \lambda k.k(\lambda x.[\![e]\!]) \\
[\![e_0 e_1]\!] &= \lambda k.[\![e_0]\!](\lambda v_0.v_0[\![e_1]\!]k)
\end{aligned}
$$

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

$$
\begin{aligned}
[\![x]\!] &= \lambda k.kx \\
[\![\lambda x.e]\!] &= \lambda k.k(\lambda x.[\![e]\!]) \\
[\![e_0 e_1]\!] &= \lambda k.[\![e_0]\!](\lambda v_0.[\![e_1]\!](\lambda v_1.v_0 v_1 k))
\end{aligned}
$$

Call-by-value CPS transformation "with continuations fi rst" (Fischer)

$$
\begin{aligned}
[\![x]\!] &= \lambda k.kx \\
[\![\lambda x.e]\!] &= \lambda k.k(\lambda k.\lambda x.[\![e]\!]k) \\
[\![e_0 e_1]\!] &= \lambda k.[\![e_0]\!](\lambda v_0.[\![e_1]\!](\lambda v_1.v_0 k v_1))
\end{aligned}
$$

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)

$$
\begin{tikzcd}
\Lambda \arrow[r, "\text{thunk transf.}"] \arrow[dr, swap, "\substack{\text{CBN}\\ \text{CPS transf.}}"] & {} \arrow[d, "\substack{\text{CBV}\\ \text{CPS transf.}}"] \\
 & {}
\end{tikzcd}
$$

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:

$$
\begin{aligned}
&[\![\lambda x.x\,x]\!] \\
&= \lambda k.k\,\lambda x.\lambda k.(\lambda k.k\,x) \\
&\qquad\qquad\qquad (\lambda v_0.(\lambda k.k\,x)\,(\lambda v_1.v_0\,v_1\,k))
\end{aligned}
$$

$$
\begin{tikzcd}
\mathrm{DS} \arrow[loop left, "\begin{matrix}\text{actual}\\ \text{red.}\end{matrix}"] \arrow[r, "\begin{matrix}\text{CPS}\\ \text{transf.}\end{matrix}"] & \mathrm{CPS} \arrow[loop right, "\begin{matrix}\text{actual + adm.}\\ \text{red.}\end{matrix}"]
\end{tikzcd}
$$

The “one-pass” approach

A CPS transformation performing

administrative reductions “at transformation time”

(Appel, Danvy and Filinski, Wand, ca. 1990)

$$\begin{tikzcd}
\mathrm{DS} \arrow[loop left, "\begin{array}{c}\text{actual}\\ \text{red.}\end{array}"] \arrow[r, "\text{CPS transf.}", "+\ \text{adm. red.}"'] & \mathrm{CPS} \arrow[loop right, "\begin{array}{c}\text{actual}\\ \text{red.}\end{array}"]
\end{tikzcd}$$

The direct-style transformation

$$\mathrm{DS} \underset{\text{DS transf.}}{\overset{\text{CPS transf.}}{\rightleftarrows}} \mathrm{CPS}$$

- 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)

$$
\begin{tikzcd}
\begin{array}{c}\text{DS +}\\ \text{call/cc}\end{array} \arrow[rr, "\text{CPS transf.}"] & & \text{CPS} \arrow[d, "\text{Normalization}"'] \\
\begin{array}{c}\text{DS +}\\ \text{call/cc}\end{array} & & \text{CPS} \arrow[ll, "\text{DS transf.}"]
\end{tikzcd}
$$

Delimited continuations

Idea: to define the rest of a $\boxed{\text{sub}}$-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 $[x_1, ..., x_n]$ 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 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.