From Calculus to Computation, Part I

Olivier Danvy

Department of Computer Science
Aarhus University
Representation

- Magritte: "This is not a pipe."

- Functional programs.
Representation

- Obiwan Kenobi: “That’s no moon.”

- Functional programs.
No self-representation, though
Toolbox (1/2)

1. closure conversion
2. CPS transformation
3. defunctionalization
Toolbox (2/2)

1. closure conversion / closure unconversion
2. CPS transformation / DS transformation
3. defunctionalization / refunctionalization

...and their left inverses.
Closure conversion

Represent each $\lambda$-abstraction with a pair:

- code, and

- environment.
The virtue of ‘closures’

- In computer science: Landin, 1964
- In mathematical logic: Hasenjaeger, 1950’s
The virtue of ‘closures’

- In computer science: Landin, 1964
- In mathematical logic: Hasenjaeger, 1950’s
  (Scholz & Hasenjaeger, Grundzüge der Mathematischen Logik, Springer 1961)
CPS transformation

1. Names intermediate results.
2. Sequentializes their computation.
3. Introduces first-class functions (continuations).
Subplan

• S
• fib
• map
A simple example (1/3)

\[ f \ x \ (g \ x) \]
A simple example (2/3)

\[
f \ x \ (g \ x)
\]

\[
\text{let } v1 = f \ x \\
v2 = g \ x \\
v3 = v1 \ v2 \\
in \ v3
\]
A simple example (3/3)

\[ f \, x \, (g \, x) \]

\[ \text{let } v_1 = f \, x \quad \lambda k. f \, x \,(\lambda v_1. \]
\[ v_2 = g \, x \quad \lambda v_2. \]
\[ v_3 = v_1 \, v_2 \quad \lambda v_3. \]
\[ \text{in } v_3 \quad \lambda k. v_3) )) \]
Subplan

• S √

• fib

• map
The Fibonacci function (1/3)

\[
\text{fib } n \\
= \text{ if } n \leq 1 \\
\text{ then } n \\
\text{ else fib}(n - 1) + \text{fib}(n - 2)
\]
The Fibonacci function (2/3)

\[
\text{fib } n \\
= \text{ if } n \leq 1 \text{ then } n
\]

\[
\text{else let } v1 = \text{ fib}(n - 1) \text{ and } v2 = \text{ fib}(n - 2) \text{ in } v1 + v2
\]
The Fibonacci function (3/3)

\[
\text{fib} \ (n, \ k) \\
= \ \text{if} \ n \leq 1 \\
\quad \text{then} \ k \ n \\
\quad \text{else} \ \text{fib}(n - 1, \ v1. \\
\quad \text{fib}(n - 2, \ v2. \\
\quad k \ (v1 + v2)))
\]
The Fibonacci function (4/3)

\[
\text{fib } n = \text{let } b = n \leq 1 \\
\text{in if } b \text{ then } n \\
\text{else let } n1 = n - 1 \\
\quad v1 = \text{fib } n1 \\
n2 = n - 2 \\
\quad v2 = \text{fib } n2 \\
\text{in } v1 + v2
\]
Subplan

• S √
• fib √
• map
The map function (1/3)

fun map (f, nil)
    = nil
| map (f, x :: xs)
    = (f x) :: (map (f, xs))
fun map (f, nil)
    = nil
  | map (f, x :: xs)
    = let v1 = f x
        v2 = map (f, xs)
     in v1 :: v2
fun map (f, nil, k) = k nil

| map (f, x :: xs, k) = f (x, \v1.
    map (f, xs, \v2.
        k (v1 :: v2)))
Subplan

- S √
- fib √
- map √
Toolbox

1. closure conversion ✓
2. CPS transformation ✓
3. defunctionalization
Defunctionalization
(a change of representation)

• Enumerate inhabitants of function space.

• Represent the function space as a sum type and a dispatching apply function.

• Transform function declarations / applications into sum constructions / calls to apply.
N.B. Closure conversion, revisited

• A special case of defunctionalization.
• Only one summand.
• Apply function inlined.
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)
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
|
fun fac (n, k)
  = fac (n - 1, fn v => k (n * v))

fun main n
  = fac (n, fn a => a)
The consumers

\[(\text{fac} : \text{int} \times (\text{int} \to \text{int}) \to \text{int})\]

\[\text{fun} \ \text{fac} \ (0, k) = k 1\]
\[\text{fun} \ \text{fac} \ (n, k) = \text{fac} \ (n - 1, \text{fn} \ v \Rightarrow k (n * v))\]

\[\text{fun} \ \text{main} \ n = \text{fac} \ (n, \text{fn} \ a \Rightarrow a)\]
The defunctionalized continuation

datatype cont = C0  
              | C1 of cont * int

fun apply_cont C0 
    = (fn a => a)  
    | apply_cont (C1 (k, n))  
      = (fn v => apply_cont (k, n * v))
Uncurried version

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

fun apply_cont (C0, a)
    = a

| apply_cont (C1 (k, n), v)
  = apply_cont (k, n * v)
```
Factorial in CPS, defunctionalized

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

fun main n = fac (n, C0)
Toolbox

1. closure conversion √
2. CPS transformation √
3. defunctionalization √
The functional correspondence

In essence:

1. closure conversion
2. CPS transformation
3. defunctionalization

On with the exercises!
The thesis

\[\lambda\text{-calculus with expl. subst. + red. strategy}\]

\[
\begin{array}{c}
\text{\textquoteleft syntactic\textquoteleft correspondence} \\
\text{abstract machine with environment}
\end{array}
\]

\[
\begin{array}{c}
\text{\textquoteleft functional\textquoteleft correspondence} \\
\text{evaluation function with environment}
\end{array}
\]
A “Scott-Tarski” evaluator
written in the syntax of Standard ML

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

datatype value =
    FUN of value -> 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)  
  = apply (eval (t0, e),  
          eval (t1, e))

and apply (FUN f, a)  
  = f a

fun main t (* : term -> value *)  
  = eval (t, nil)
John Reynolds’s question

Does this interpreter define

- a call-by-name language, or
- a call-by-value language?
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) 
    = apply (eval (t0, e), 
                    eval (t1, e)) 
and apply (FUN f, a) 
    = f a
John Reynolds’s point

Be mindful of the evaluation order of the meta-language:

- Call by name yields call by name.
- Call by value yields call by value.
Well-defined definitional interpreters

- Evaluation-order independent.
- First-order.
Closure conversion of the def. int.

datatype value = FUN of \( \text{term} \times \text{env} \)

with

\( \text{env} = \text{value list} \)

(* main : \text{term} \rightarrow \text{value} *)

fun main t

= eval (t, nil)
and eval (IND n, e) = List.nth (e, n)
  | eval (ABS t, e) = FUN (t, e)
  | eval (APP (t0, t1), e) = apply (eval (t0, e),
                               eval (t1, e))

and apply (FUN (t, e), a) = eval (t, a :: e)
CPS transformation of the def. int.

datatype value = FUN of term * env
with
    type env = value list

    type ans = value
    type cont = value -> ans

(* main : term -> ans *)
fun main t
  = eval (t, nil, fn v => v)
and eval (IND n, e, k)
  = k (List.nth (e, n))
| eval (ABS t, e, k)
  = k (FUN (t, e))
| eval (APP (t0, t1), e, k)
  = eval (t0, e, fn v0 =>
            eval (t1, e, fn v1 =>
                apply (v0, v1, k)))
and apply (FUN (t, e), a, k)
  = eval (t, a :: e, k)
Defunctionalization of the def. int.

datatype value = FUN of term * env
with
type env = value list
and ans = value

datatype cont =
    C2 of term * env * cont
| C1 of denval * cont
| C0
fun main t = eval (t, nil, C0)

and apply_cont (C2 (t1, e, k), v0) = eval (t1, e, C1 (v0, k))
| apply_cont (C1 (v0, k), v1) = apply (v0, v1, k)
| apply_cont (C0, v) = v
and eval (IND n, e, k)
    = apply_cont (k, List.nth (e, n))
| eval (ABS t, e, k)
    = apply_cont (k, FUN (t, e))
| eval (APP (t0, t1), e, k)
    = eval (t0, e, C2 (t1, e, k))

and apply (FUN (t, e), a, k)
    = eval (t, a :: e, k)
“Machine-like character”

Reynolds: see the “machine-like character”
of this interpreter?
In summary

evaluator for $\lambda$-terms

- closure conversion
- CPS transformation
- defunctionalization

an abstract machine