From Calculus to Computation, Part II

Olivier Danvy

Department of Computer Science

Aarhus University

The thesis

λ-calculus with expl. subst. + red. strategy

↓ 'syntactic' correspondence

abstract machine with environment

↑ 'functional' correspondence

evaluation function with environment

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 term * env
withtype   env = value list

(*  main : term -> 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
withtype   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
withtype   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 λ-terms

$\downarrow$

closure conversion

CPS transformation

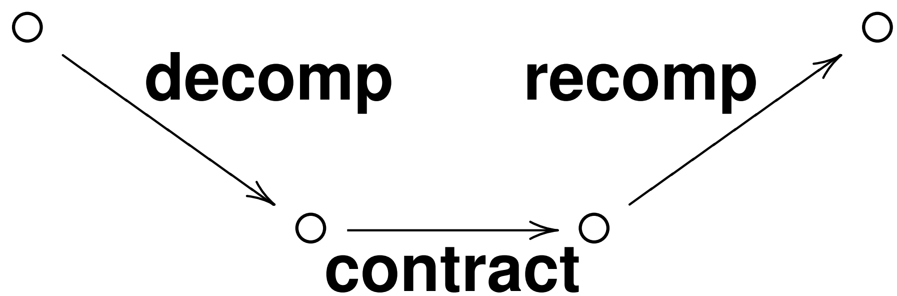
defunctionalization

an abstract machine

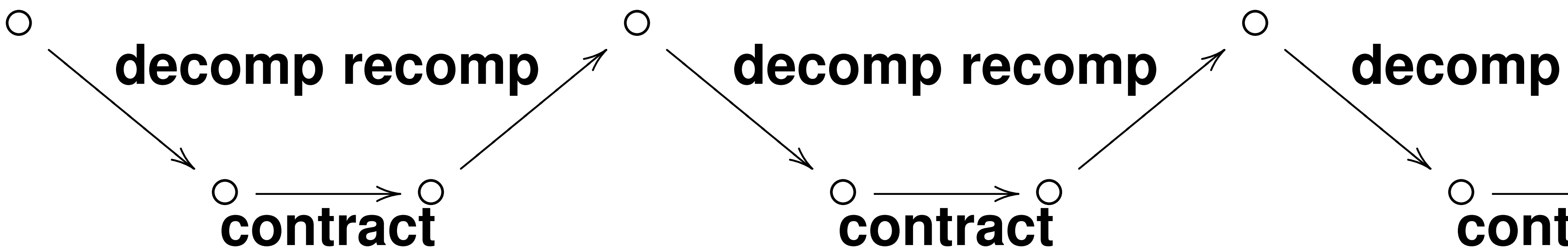
Refocusing

- One-step reduction.
- Reduction-based evaluation.

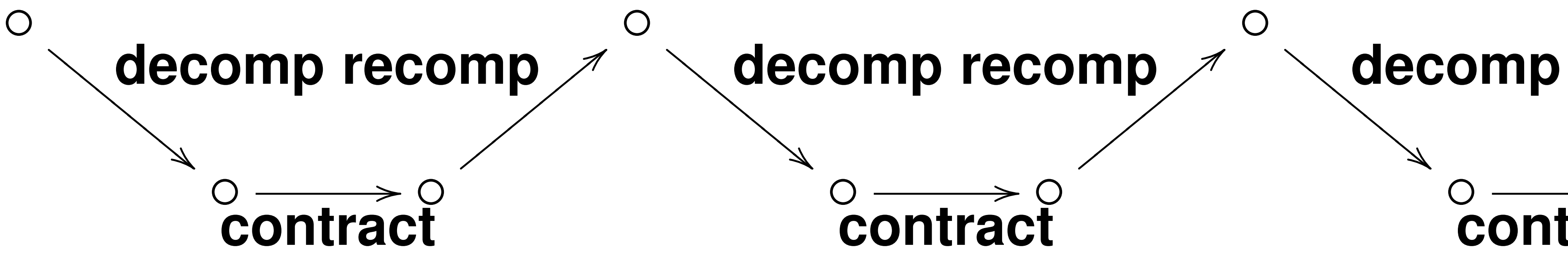
One-step reduction visually



Reduction-based normalisation visually

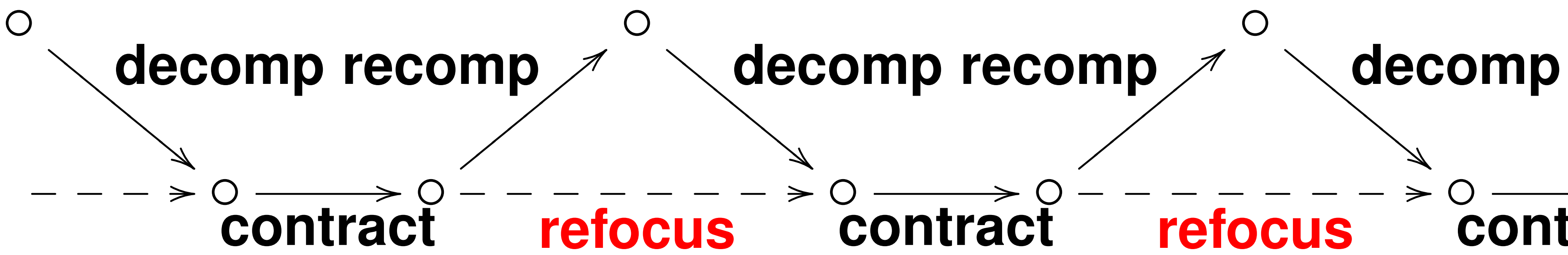


Reduction-based normalisation visually



A case for deforestation (to a man with a hammer).

Refocusing



decomp recomp
decomp recomp
decomp
contract
refocus
contract
refocus
cont