Continuations & Co-exponentials

Vikraman Choudhury
University of Glasgow

LFCS Seminar, University of Edinburgh
May 9, 2023

Last updated on May 10, 2023 at 12:03

Currying

We all know these:

```
curry :: ((a, b) → c) → a → (b → c)
curry f a b = f (a, b)

uncurry :: (a → (b → c)) → (a, b) → c
uncurry f (a, b) = f a b
```

Currying and Co-currying?

We all know these:

```
curry :: ((a, b) → c) → a → (b → c)
curry f a b = f (a, b)

uncurry :: (a → (b → c)) → (a, b) → c
uncurry f (a, b) = f a b
```

Puzzle: can we dualize these?

```
cocurry :: (c → (a + b)) → (c - b) → a
councurry :: ((c - b) → a) → (c → (a + b))
```

No go

Technical results from category theory say you can't!

If you could (by LAPC/RAPL):

$$A \times 0 \cong 0 \qquad\qquad A + 1 \cong 1$$

If these were propositions in logic:

$$a \wedge \bot \iff \bot \qquad\qquad a \vee \top \iff \top$$

But as types in a programming language, you'd have:

$$2 \cong 1 + 1 \cong 1$$

You can have a logic with subtraction, but not a programming language!

No go

- ⋆ Boileau & Joyal: A cartesian closed and co-cartesian co-closed category is a preorder.
- ⋆ Abramsky: A $*$-autonomous category in which the monoidal structure is cartesian is a preorder.

Linear logic comes to the rescue...

- ⋆ Crolard: Subtractive logic
- ⋆ Eades, Bellin: Co-intuitionistic Adjoint Logic
- ⋆ Abramsky: connection between limitative results in proof theory and No-Go theorems in quantum mechanics

But wait, I will show you a magic trick...

Currying and Co-currying

My kingdom for a horse…

```
curry :: ((a, b) → c) → a → (b → c)
curry f a b = f (a, b)

uncurry :: (a → (b → c)) → (a, b) → c
uncurry f (a, b) = f a b

cocurry :: (c ⇒ (a + b)) → (c - a) ⇒ b
cocurry f (c, k1) = Cont $ \k2 → runCont (f c) (either k1 k2)

councurry :: ((c - a) ⇒ b) → (c ⇒ (a + b))
councurry f c = Cont $ \k → runCont (f (c, k . Left)) (k . Right)
```

I snuck in two kinds of arrows: →, ⇒, but what is `c - a`?

Continuations

From Reynolds (1993):

> «...settings in which continuations were found useful: They underlie a method of program transformation (into continuation-passing style), a style of definitional interpreter (defining one language by an interpreter written in another language), and a style of denotational semantics (in the sense of Scott and Strachey). In each of these settings, by representing "the meaning of the rest of the program" as a function or procedure, continuations provide an elegant description of a variety of language constructs, including call by value and goto statements.»

From Matt Might's blog:

> «...they're always explained with quasi-metaphysical phrases: "time travel", "parallel universes", "the future of the computation".»

Continuation-Passing Style

How I learned continuations in Dan Friedman's C311:

```
(define factorial
  (lambda (n)
    (cond
      [(zero? n) 1]
      [else (* n (factorial (sub1 n)))])))
```

This program isn't tail-recursive!

Continuations to the rescue…

Continuation-Passing Style

We can transform this into CPS:

```
(define factorial-cps
  (lambda (n k)
    (cond
      [(zero? n) (k 1)]
      [else (factorial-cps (sub1 n)
                           (lambda (v) (k (* n v))))])))

  (define factorial
    (lambda (n)
      (factorial-cps n (lambda (v) v))))
```

Delimited Continuations

Types help you see what's going on...

```
factorialCPS :: Int → (Int → r) → r
factorialCPS n k =
  if n == 0
    then k 1
    else factorialCPS (n - 1) $ \v → k (n * v)

factorial :: Int → Int
factorial n = factorialCPS n $ \v → v
```

Continuations are encoded as functions: $a \to r$.

Continuation monad

Monads make this even better!

```
newtype Cont r a = Cont { runCont :: (a → r) → r }

return :: a → Cont r a
return a = Cont $ \k → k a

(>>=) :: Cont r a → (a → Cont r b) → Cont r b
Cont g >>= f = Cont $ \k2 → g $ \a → runCont (f a) k2
```

Now rewrite `facrorialCPS` using the continuation monad…

Continuation monad

Using do notation:

```
factorialCont :: Int → Cont r Int
factorialCont n =
  if n == 0
    then return 1
    else do
      v ← factorialCont (n - 1)
      return (n * v)

factorial :: Int → Int
factorial n = runCont (factorialCont n) $ \v → v
```

This is automatically tail-recursive!

This is CPS without explicitly thinking about continuations as functions.

CPS, formally

There are many ways of formalising CPS:

- Plotkin-style CPS

  `a → b` turns into `a → (b → r) → r`, or `a → Cont r b`.

- Fischer-style CPS

  `a → b` turns into `(b → r) → (a → r)`.

There are several CPS calculi and connections to classical logic.

Embrace these ideas and take a step further…

Co-exponentials

Allow me to write:

- `a* = a → r`
  – a continuation for `a`, or
  – a handler for `a`

- `b - a = (b, a*)`
  – a value of `b`, with a handler for `a`, or
  – a value of `b`, with a typed hole for `a`

- `a ⇒ b = a → Cont r b`
  – a CPS transformed function `a → b`

Now I'll reveal the trick…

Co-exponentials

This is co-currying with subtraction and ⇒:

```
cocurry :: (c ⇒ (a + b)) → (c - a) ⇒ b
cocurry f (c, k1) = Cont $ \k2 →
  runCont (f c) $ \case
    Left a → k1 a
    Right b → k2 b

councurry :: ((c - a) ⇒ b) → (c ⇒ (a + b))
councurry f c = Cont $ \k →
  let k1 = k . Left
      k2 = k . Right
   in runCont (f (c, k1)) k2
```

Co-exponentials

This is co-currying with all the explicit types:

```
cocurry :: (c → Cont r (a + b)) → (c, a → r) → Cont r b
cocurry f (c, k1) = Cont $ \k2 →
  runCont (f c) $ \case
    Left a → k1 a
    Right b → k2 b

councurry :: ((c, a → r) → Cont r b) → (c → Cont r (a + b))
councurry f c = Cont $ \k →
  let k1 :: a → r
      k1 = k . Left
      k2 :: b → r
      k2 = k . Right
   in runCont (f (c, k1)) k2
```

Co-exponentials

It computes this isomorphism…

```
  c → ((a + b) → r) → r
≅ c → (a → r, b → r) → r
≅ c → (a → r) → (b → r) → r
≅ (c, a → r) → (b → r) → r
```

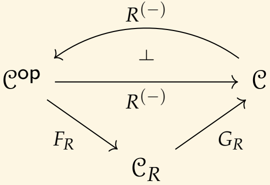
From left to right, it splits a continuation for `a + b`.

From right to left, it joins two continuations for `a` and `b`.

You can implement these in your favorite programming language if you have currying and sums.

A micrological study of continuations

There is an elegant mathematical theory behind all of this.



The Kleisli category of the continuation monad is co-cartesian co-closed!

It's a miraculous adjunction:

$$(-) \times R^X \dashv X + (-)$$

This fact (in the dual sense) was known to several experts since the 90s (see **slide 44**), but it is underappreciated and seems to have been forgotten.

I try to explain this in a more conceptual way (see **slide 47**).

Co-exponential operators

You can implement these operators in your favorite programming language if you have currying and sums.

Currying gives you `eval` and `uneval` (higher-order pairing).

```
id :: a → a
id a = a

eval :: (a → b, a) → b
eval = curry id

uneval :: a → (b → (a, b))
uneval = uncurry id
```

Co-exponential operators

Dually, co-currying gives you `coeval` and `couneval`.

```
idk :: a ⇒ a
idk = return

coeval :: b ⇒ (a + (b - a))
coeval = councurry idk

couneval :: ((a + b) - a) ⇒ b
couneval = cocurry idk
```

`coeval` *creates a choice*, `couneval` *annihilates a choice.*

Compare: law of excluded middle: $\cdot \vdash a + a^*$

Compare: creation/annihilation operators in differential LL (C., Fiore).

Co-lambda and Co-application

Let's simplify these into simpler combinators...

```
colam :: (a* ⇒ b) ⇒ (() ⇒ (a + b))
colam f = councurry (f . snd)

coapp :: (() ⇒ (a + b), a*) ⇒ b
coapp (f, k1) = f () >>= couneval . (,k1)
```

No more → arrows, now I can work with ⇒ arrows directly.

I will extend Moggi's computational metalanguage with these two operators.

λ*

Start from a call-by-value lambda calculus.

Add sum types, A^*, and two typing rules…

Binding a value gives you a function!

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).e : A \Rightarrow B} \qquad \frac{\Gamma \vdash e_1 : A \Rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\frac{\Gamma, x : A^* \vdash e : B}{\Gamma \vdash \widetilde{\lambda}(x : A^*).e : A + B} \qquad \frac{\Gamma \vdash e_1 : A + B \qquad \Gamma \vdash e_2 : A^*}{\Gamma \vdash e_1 \widetilde{\ } e_2 : B}$$

Binding a continuation gives you a choice!

λ*

And two call-by-value equations...

$$\frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A).e)\ v \equiv e[v/x] : A \Rightarrow B} \qquad \frac{\Gamma \vdash v : A \Rightarrow B}{\Gamma \vdash \lambda(x : A).v\ x \equiv v : A \Rightarrow B}$$

$$\frac{\Gamma, x : A^* \vdash e : B \qquad \Gamma \vdash v : A^*}{\Gamma \vdash (\widetilde{\lambda}(x : A^*).e)\ v \equiv e[v/x] : B} \qquad \frac{\Gamma \vdash v : A + B}{\Gamma \vdash \widetilde{\lambda}(x : A^*).\widetilde{v\ x} \equiv v : A + B}$$

Or, Freyd categories with Kleisli exponentials & co-exponentials (see **slide 51**):

$$\mathcal{C}(J(C \times A), B) \cong \mathcal{V}(C, A \Rightarrow B) \qquad \mathcal{C}(A^* \cdot B, C) \cong \mathcal{C}(B, J(A) + C)$$

This is a fine-grained language for understanding control flow using continuations under the hood.

λ^*

Key ideas

- *Main trick*: Split values and computations (double negations).
- You can't create continuations using functions, only co-exponentials.
- No need to split contexts, and no polarities necessary.

Semantics

- It admits weakening and substitution.
- It has operational, categorical, and adequate denotational semantics.
- It is a conservative extension of STLC.
- Axiomatized by closed co-closed Freyd categories.

Applications

- Combines exponentials and co-exponentials, but is *not degenerate*.
- Clean encoding of subtractive/co-intuitionistic logics: ${}^{B}A = B \times A^*$.
- Clean language of values and continuations (cf. $\mu\tilde{\mu}$, $\lambda\mu$, polarities)

Philosophical Musings

Magic tricks are surprising, but once you reveal the trick, they become boring.

What lessons did we learn from this trick?

- No-go theorems
  - Trick to getting around them: splitting values and computations.
  - We turned products into premonoidal products.
  - These are well-known techniques in PL.
  - Instead of a *programming language*, we get a *call-by-value* programming language.
  - Where else can we play this game?

Philosophical Musings

What lessons did we learn from this trick?

- Duality
  - There is a deep duality between functions and continuations.
  - Therefore, they should enjoy the same ontological status.
  - We shouldn't conflate continuations with functions.
  - Co-exponentials are a powerful interface, as we will see next.
  - Duality is a fashionable trend in PL:

(pairs) products	co-products (sums)
(effects) monads	co-monads (co-effects, purity)
(induction) initial algebras	final co-algebras (co-induction)
(functions) exponentials	co-exponentials (continuations)

Co-exponentials in Action

- ⋆ Classical Logic & Control Operators
- ⋆ Speculative Execution & Backtracking
- ⋆ Effect Handlers
- ⋆ First-order Control Flow

Programming in λ^* is like programming in Haskell with monadic operations and two operators: `colam`, `coapp`.

Classical logic and control

I can derive classical logic and control operators.

The identity co-function: $\tilde{\lambda}(x : A^*) \,.\, x$ gives you LEM!

```
lem :: a + a*
lem = colam idk
```

`callCC` comes from `colam`!

```
codiag :: a + a → a
codiag = either id id

callCC :: (a* ⇒ a) ⇒ a
callCC = fmap codiag . colam
```

Backtracking operators

A toy DSL for backtracking using co-exponentials in Haskell…

```
assumeRight :: ((a → r) → Cont r b) → Cont r (a + b)
assumeRight = colam

resolveRight :: Cont r (a + b) → (a → r) → Cont r b
resolveRight = coapp
```

A way to swap choices…

```
swap :: (a + b) → (b + a)
swap = either Right Left
```

Compare: Thielecke's Double-Barrelled CPS

Backtracking operators

Some derived operators:

```
assumeLeft :: ((b → r) → Cont r a) → Cont r (a + b)
assumeLeft = fmap swap . colam

resolveLeft :: Cont r (a + b) → (b → r) → Cont r a
resolveLeft = coapp . fmap swap

assumeBoth :: ((a → r) → (b → r) → r) → Cont r (a + b)
assumeBoth f = assumeRight $ \k1 → cont $ \k2 → f k1 k2

resolveBoth :: Cont r (a + b) → (a → r) → (b → r) → r
resolveBoth f k1 = runCont (resolveRight f k1)
```

Backtracking SAT solver

```
data Prop = PVar String | PZero | POne
          | PAnd Prop Prop | POr Prop Prop | PNot Prop

solve :: Env Bool → Prop → Cont r (Fail + Succ r)
solve env phi =
  case phi of
    PZero →
      assumeLeft $ \succ →
        return ()
    POne →
      assumeRight $ \fail →
    ...
```

Demo?

Compare: Jacob Errington's SAT solver, Jules Hedges' SAT solver.

Speculative Execution & Backtracking

You want to write a program of type `a + b`...

- Speculative Execution
  - You need to make a choice `a + b`, but you can't commit to a choice `Left` or `Right`.
  - Speculatively, choose `b` with `assumeRight`. Then, `assumeRight` gives you a free continuation `a*`. You may or may not use it.
  - Do some computation and produce `b`.

Speculative Execution & Backtracking

The user of your `a + b` program wants to execute it...

- **Backtracking**
  - There are two ways to use these sum types: `case` or `resolve`.
  - If they case on the sum, there are two execution paths:
    - ⋆ When they use `Right b`, they execute your computation.
    - ⋆ When they use `Left a`, the system jumps to a top-level continuation.

Speculative Execution & Backtracking

- Backtracking
  - If they use a `resolve` combinator:
    - ⋆ If they call `resolveRight`, they have to plugin a continuation `a`*, producing `b`. This continuation gets passed in to the environment of the original computation.
    - ⋆ If they call `resolveLeft`, they have to plugin a continuation `b`*, and they get an `a`. This continuation gets spliced into the top-level stack.

Key idea: *two continuations* for *two execution paths.*

All this can be translated to $\lambda*$, and the equations of $\lambda*$ validate these informal ideas of speculative execution and backtracking. This is an *algebraic axiomatization of control effects and handlers.*

Effect handlers

I can derive effect handlers using co-exponential operators.

Well-known to Haskellers: Church-encode the free monad...

```
newtype Free f a = Free { runFree :: forall r. (f r → r) → Cont r a }
```

There are two continuations to manage: the handler (algebra) `f r → r`, and the generator `a → r`.

```
colamFree :: Free f a → Cont r (f r + a)
colamFree f = colam $ \alg → cont $ \gen →
  runCont (runFree f alg) gen

foldFree :: Functor f ⇒ (f r → r) → (a → r) → Free f a → r
foldFree alg gen = reset0 . fmap (either alg gen) . colamFree
```

Demo?

Whither functions?

We've been using higher-order functions to encode continuations.

Do we need to?

Some ideas:

- Kleisli exponentials

  From the point of view of Freyd categories:

  We don't need $\mathcal{V}$ to be cartesian closed, we only need Kleisli exponentials.

  But in practice, $\mathcal{V}$ is cartesian closed, with a strong monad.

Whither functions?

- Classical encoding

Encode functions $A \to B$ as $B + A^*$.

This gives a CPS-ed function:

$$C \to (B + A^*) \cong C \times B^* \to A^*$$

...which is a compromise.

Whither functions?

- First-order languages with co-exponentials

Instead, what if we had a first-order language, and added co-exponentials?

Hasegawa's trick: using functional completeness, split λ-calculus into two first-order calculi: κ and ζ-calculi. This is like an arrow calculus.

Using co-exponentials, I can dualise functional completeness and produce a first-order arrow language with control flow.

Functional Completeness

Functional Completeness

STLC/CCCs enjoy a functional completeness property (Lambek & Scott 1986), like the deduction theorem in proof theory.

- to prove $A \to B$, it is sufficient to prove B assuming A.
- to write a program of type $A \to B$, it is sufficient to write a program of type B, assuming a free variable of type A.

Dual of Functional Completeness

CoCCoCCs enjoy a dual of functional completeness (interpreting co-exponential objects using continuations):

- to prove $A + B$, it is sufficient to prove B assuming A^*.
- to write a program of type $A + B$, it is sufficient to write a program of type B, assuming a free continuation for A.

This can be proved by abstract nonsense (see **slide 52**).

κ/ζ

Hasegawa splits λ-calculus into κ/lift and ζ/pass: these are arrow calculi, arrows have identity and composition, and these operators.

$$\frac{\Gamma \vdash c : 1 \rightsquigarrow C}{\Gamma \vdash \mathsf{lift}_A(c) : A \rightsquigarrow C \times A} \qquad \frac{\Gamma, x : 1 \rightsquigarrow C \vdash f : A \rightsquigarrow B}{\Gamma \vdash \kappa x^C.f : C \times A \rightsquigarrow B}$$

$$\frac{\Gamma \vdash c : 1 \rightsquigarrow C}{\Gamma \vdash \mathsf{pass}_B(c) : (C \Rightarrow B) \rightsquigarrow B} \qquad \frac{\Gamma, x : 1 \rightsquigarrow C \vdash f : A \rightsquigarrow B}{\Gamma \vdash \zeta x^C.f : A \rightsquigarrow (C \Rightarrow B)}$$

Equational theory on **slide 53**.

κ^*/ζ^*

Dualising...

$$\frac{\Gamma \vdash c : 1 \rightsquigarrow C^*}{\Gamma \vdash \mathsf{lift}^*_A(c) : A \rightsquigarrow (A - C)} \qquad \frac{\Gamma, x : 1 \rightsquigarrow C^* \vdash f : A \rightsquigarrow B}{\Gamma \vdash \kappa^* x^C.f : (A - C) \rightsquigarrow B}$$

$$\frac{\Gamma \vdash c : 1 \rightsquigarrow C^*}{\Gamma \vdash \mathsf{pass}^*_B(c) : (C + B) \rightsquigarrow B} \qquad \frac{\Gamma, x : 1 \rightsquigarrow C^* \vdash f : A \rightsquigarrow B}{\Gamma \vdash \zeta^* x^C.f : A \rightsquigarrow (C + B)}$$

This gives you a first-order programming language with control flow operators.

If you add natural numbers, you get (first-order) primitive recursion with control flow. What is its expressive power? Can you write `genericcount`/`effcount`?

Equational theory on **slide 54**.

Programming in κ^*/ζ^*

These operators allow you to do surgery on first-order programs.

With an indeterminate $z : Z^*$,

$$A \xrightarrow{\mathsf{lift}^*(z)} A - Z \xrightarrow{\zeta^* z.id} Z + (A - Z) \xrightarrow{Z + \kappa^* z.id} Z + A \xrightarrow{\mathsf{pass}^*(z)} A$$

$$A \xrightarrow{\zeta^* z.id} Z + A \xrightarrow{Z + \mathsf{lift}^*(z)} Z + (A - Z) \xrightarrow{\mathsf{pass}^*(z)} A - Z \xrightarrow{\kappa^* z.id} A$$

Rewrite programs using ζ^*/pass^*:

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D \xrightarrow{e} E$$

$$A \xrightarrow{f} B \xrightarrow{\zeta^* z.g} Z + C \xrightarrow{h'} Z + D \xrightarrow{\mathsf{pass}^*(z)} D \xrightarrow{e} E$$

A mechanism for breakpoints, checkpoints, code pointers, debugging?

Some type isomorphisms

Like Tarski's high-school algebra identities, but with subtraction:

$$
\begin{aligned}
X - 0 &\cong X \\
0 - X &\cong 0 \\
(X + Y) - Z &\cong (X - Z) + (Y - Z) \\
(X + Z) \Rightarrow Y &\cong (Y - X) \Rightarrow Z
\end{aligned}
$$

These make more sense once you translate them back to STLC with an R.

Lawvere's ∂ operator

Examples of co-Heyting algebras in nature:

- ⋆ Closed subsets of a topological space
- ⋆ Subobject lattices of presheaf categories

Following Lawvere, define the boundary operator: $\partial A = A - A$.

These Leibniz maps exist:

$$\partial(A \times B) \to \partial A \times B + A \times \partial B$$

$$\partial A \times B + A \times \partial B \to \partial(A \times B)$$

To make this an iso, however, Lawvere requires a de Morgan law:

$$(A \times B)^* \cong A^* + B^*$$

Session Types

I discovered these when studying session types & classical linear logic using (strict) star-autonomous categories, following Mellies' articles on negation, dialogue categories, chiralities, tensorial logic.

A star-autonomous category is linearly-distributive with appropriate duals.

$$A \otimes (-) \dashv A^* \parr (-) \qquad\qquad (-) \otimes B^* \dashv (-) \parr B$$

This gives: $A \multimap B = A^* \parr B$, and $A \multimapinv B = A \otimes B^*$.

Cut in (H)CP is:

$$(B \multimap C) \otimes (A \multimap B) \longrightarrow (A \multimap C)$$

Dually:

$$(B \multimapinv C) \parr (A \multimapinv B) \longleftarrow (A \multimapinv C)$$

Co-exponentials in disguise

Some places where co-exponentials appear:

- ⋆ CBV translation of $\mu\tilde{\mu}$ calculus
- ⋆ Streicher, Reus, Hofmann: Semantics of $\lambda\mu$ calculus
- ⋆ Thielecke's thesis: Section 4.5
- ⋆ Selinger's co-control categories

Note: We fixed the result type R, but we can do more if we choose different Rs, e.g. $\Omega^{(-)} : \mathcal{E}^{\mathsf{op}} \to \mathcal{E}$ is monadic.

Conclusion

Duality

Higher-order functions give you exponentials.

Higher-order continuations give you co-exponentials

Co-exponential operators

- Algebraic axiomatization of control flow using continuations
- Interpretation of bi-intuitionistic, subtractive, classical logic
- Backtracking and Control operators
- Fine-grained study of effect handlers

Decomposing functions

- Linear logic gives $A \to B = \,!A \multimap B$ and $A \multimap B = A^* \parr B$.
- Girardian comonads and Moggi's monads give: $DA \to TB$.
- Continuations/co-exponentials give: $A \to B = A^* + B$.

Bonus slides

A micrological study of continuations

Start with a cartesian closed category $\mathcal{C}$ with a fixed object R. Since it is self-enriched, we can write Y^X for the hom $\mathcal{C}(X, Y)$.

$$\mathcal{C}^{\mathsf{op}} \underset{R^{(-)}}{\overset{R^{(-)}}{\rightleftarrows}} \mathcal{C} \quad (\bot)$$

The contravariant negation functor is strong self-adjoint on the left.

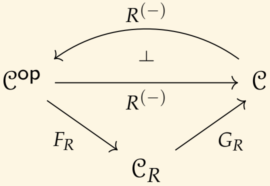
$$R^{(-)} : \mathcal{C}^{\mathsf{op}} \to \mathcal{C}$$
$$A \mapsto R^A$$
$$B \xrightarrow{f} A \mapsto R^A \xrightarrow{(-)\circ f} R^B$$

$$st_{X,Y} : \mathcal{C}(X, Y) \to \mathcal{C}^{\mathsf{op}}(R^X, R^Y)$$
$$f \mapsto R^Y \xrightarrow{(-)\circ f} R^X$$

$$\mathcal{C}^{\mathsf{op}}(R^X, Y) = \mathcal{C}(Y, R^X) \cong \mathcal{C}(X \times Y, R) \cong \mathcal{C}(X, R^Y)$$

A micrological study of continuations

By (bo, ff) factorisation, $R^{(-)}$ splits as follows, $\mathcal{C}_R$ is the full-image of $R^{(-)}$.



$$\begin{aligned} F_R : \mathcal{C}^{\mathsf{op}} &\to \mathcal{C}_R \\ A &\mapsto A \\ B \xrightarrow{f} A &\mapsto R^A \xrightarrow{(-)\circ f} R^B \end{aligned}$$

$$\begin{aligned} G_R : \mathcal{C}_R &\to \mathcal{C}^{\mathsf{op}} \\ A &\mapsto R^A \\ R^A \xrightarrow{f} R^B &\mapsto R^A \xrightarrow{f} R^B \end{aligned}$$

F_R has a left-adjoint: $R^{(-)} \circ G_R \dashv F_R$.

$$\mathcal{C}^{\mathsf{op}}(R^{G_R(X)}, Y) = \mathcal{C}(R^{R^X}, Y) \cong \mathcal{C}(R^X, R^Y) = \mathcal{C}_R(X, F_R(Y))$$

A micrological study of continuations

If $\mathcal{C}$ has co-products, they become products in $\mathcal{C}^{\mathsf{op}}$, then products in $\mathcal{C}_R$.

$$\begin{aligned}\mathcal{C}_R(Z, X+Y) &= \mathcal{C}(R^Z, R^{X+Y}) \\ &\cong \mathcal{C}(R^Z, R^X \times R^Y) \\ &\cong \mathcal{C}(R^Z, R^X) \times \mathcal{C}(R^Z, R^Y) = \mathcal{C}_R(Z, X) \times \mathcal{C}_R(Z, Y)\end{aligned}$$

Since G_R is ff, it reflects limits.

$$G^R(X+Y) = R^{X+Y} \cong R^X \times R^Y = G_R(X) \times G_R(Y)$$

G_R wants to be a cartesian-closed functor.

If $X \Rightarrow Y$ was the exponential in $\mathcal{C}_R$,

$$G_R(X \Rightarrow Y) \cong G_R(Y)^{G_R(X)} = (R^Y)^{R^X} \cong R^{Y \times R^X}$$

Hence, $X \Rightarrow Y = Y \times R^X$, making G_R a cartesian-closed functor.

A micrological study of continuations

Finally, the continuation monad on $\mathcal{C}$ is $T_R = R^{(-)} \circ R^{(-)}$.

Consider the Kleisli arrows:

$$\mathcal{C}_{T_R}(X, Y) = \mathcal{C}(X, T_R(Y)) = \mathcal{C}(X, R^{R^Y}) \cong \mathcal{C}(R^Y, R^X) = \mathcal{C}_R^{\mathsf{op}}(X, Y)$$

Since $\mathcal{C}_R$ is cartesian closed, $\mathcal{C}_{T_R}$ becomes co-cartesian co-closed.

This uses self-enrichment and strength, and can be done more generally in an enriched setting.

Closed co-closed Freyd categories

A distributive closed Freyd category $\mathcal{V} \xrightarrow{J} \mathcal{C}$ has:

- ⋆ Kleisli exponentials:

  $J((-) \times A) : \mathcal{V} \to \mathcal{C}$ has a right adjoint $A \Rightarrow (-)$:
  $$\mathcal{C}(J(C \times A), B) \cong \mathcal{V}(C, A \Rightarrow B)$$

Add:

- ⋆ a function $(-)^* : |\mathcal{V}| \to |\mathcal{V}|$ on the objects of $\mathcal{V}$

- ⋆ Kleisli co-exponentials:

  $J(A) + (-) : \mathcal{C} \to \mathcal{C}$ has a specified left adjoint $A^* \cdot (-)$:
  $$\mathcal{C}(A^* \cdot B, C) \cong \mathcal{C}(B, J(A) + C)$$

This is a candidate axiomatisation of λ^*.

Functional Completeness

For a CCC $\mathcal{C}$:

- $A \times (-) : \mathcal{C} \to \mathcal{C}$ is a comonad, $(-)^A : \mathcal{C} \to \mathcal{C}$ is a monad.
- The Kleisli category $\mathcal{C}_{A\times(-)}$ is a CCC (with an indeterminate value $1 \to A$).
- $\mathcal{C}_{A\times(-)}$ and $\mathcal{C}_{(-)^A}$ are canonically equivalent, by currying.

For a CoCCCoCC $\mathcal{C}$:

- $A + (-) : \mathcal{C} \to \mathcal{C}$ is a monad, ${}^A(-) : \mathcal{C} \to \mathcal{C}$ is a comonad.
- The Kleisli category $\mathcal{C}_{A_{(-)}}$ is a CoCCCoCC (with an indeterminate continuation $1 \to A^*$).
- $\mathcal{C}_{A_{(-)}}$ and $\mathcal{C}_{A+(-)}$ are canonically equivalent, by co-currying.

Equational Theory of κ/ζ

Equational theory of κ:

$$\frac{\Gamma \vdash f : C \times A \rightsquigarrow B}{\Gamma \vdash \kappa x^C.(f \circ \mathsf{lift}_A(x)) \equiv f : C \times A \rightsquigarrow B}$$

$$\frac{\Gamma, x : 1 \rightsquigarrow C \vdash f : A \rightsquigarrow B \qquad \Gamma \vdash c : 1 \rightsquigarrow C}{\Gamma \vdash \kappa x^C.f \circ \mathsf{lift}_A(c) \equiv f[c/x] : A \rightsquigarrow B}$$

Equational theory of ζ:

$$\frac{\Gamma \vdash f : A \rightsquigarrow (C \Rightarrow B)}{\Gamma \vdash \zeta x^C.(\mathsf{pass}_B(x) \circ f) \equiv f : A \rightsquigarrow (C \Rightarrow B)}$$

$$\frac{\Gamma, x : 1 \rightsquigarrow C \vdash f : A \rightsquigarrow B \qquad \Gamma \vdash c : 1 \rightsquigarrow C}{\Gamma \vdash \mathsf{pass}_B(c) \circ \zeta x^C.f \equiv f[c/x] : A \rightsquigarrow B}$$

Equational Theory of κ^*/ζ^*

Equational theory of κ^*:

$$\frac{\Gamma \vdash f : A - C \rightsquigarrow B}{\Gamma \vdash \kappa^* x^C.(f \circ \mathsf{lift}^*_A(x)) \equiv f : (A - C) \rightsquigarrow B}$$

$$\frac{\Gamma, x : 1 \rightsquigarrow C^* \vdash f : A \rightsquigarrow B \qquad \Gamma \vdash c : 1 \rightsquigarrow C^*}{\Gamma \vdash \kappa^* x^C.f \circ \mathsf{lift}^*_A(c) \equiv f[c/x] : A \rightsquigarrow B}$$

Equational theory of ζ^*:

$$\frac{\Gamma \vdash f : A \rightsquigarrow (C + B)}{\Gamma \vdash \zeta^* x^C.(\mathsf{pass}^*_B(x) \circ f) \equiv f : A \rightsquigarrow (C + B)}$$

$$\frac{\Gamma, x : 1 \rightsquigarrow C^* \vdash f : A \rightsquigarrow B \qquad \Gamma \vdash c : 1 \rightsquigarrow C^*}{\Gamma \vdash \mathsf{pass}^*_B(c) \circ \zeta^* x^C.f \equiv f[c/x] : A \rightsquigarrow B}$$