C[ontinuations]:

continued

and

to be continued

At 関数型まつり 2025 June 14, 2025

Satoru Kawahara

Twitter絵師ハカセの質問コーナー [1]

Q. What is the most expressive concept of **functional programming**?

[1] https://arkw.net/products/web/hakase/

[*1]

Q. What is the most expressive concept of **functional programming**?

A. Definitely, it's a

CONTINUATION!

[要出典]

[*1] https://arkw.net/products/web/hakase/

Today's Topic

Learn about

Today’s Topic

Learn about

Continuations:

The Concept, History, and Practice

Continuations:
continued and to be continued

- **1. Introduction**
  - Today’s Topic
  - Continuations?
- **2. The Evolution of Continuations**
  - Dump and J operator, SECD machine
  - `call/cc`: `goto` Practical
  - Delimited Continuations
  - Continuation-Passing Style
  - Compiling with Continuations
  - Correspondence between procedural
- **3. Continuations with Effects**
  - Why continuations for effects?
  - Monads
  - Algebraic Effect Handlers
- **4. Conclusion**

Who talks

- 🏢 **eiicon, co.,ltd.**
- 📎 Nymphium
- ❤️ OCaml
- 🤓 Interested in:
  - Programming language theory and implementations
  - Control flow and its operators

こんにちは、
びしょ〜じょです。

■ Who talks

- 🏢 **eiicon, co.,ltd.**
- 📎 Nymphium
- ❤️ OCaml
- 🤓 Interested in:
  - Programming language theory and implementations
  - Control flow and its operators
- 💪 Motto:

継続は力なり

Continuation is power

こんにちは、
びしょ〜じょです。

Continuations?

Do you *remember* continuations?

Continuations?

Do you *remember* continuations?

🙋‍♂️ Hmm. . . . . . I forget. Help me remember!

●Continuations?

Do you *remember* continuations?

- 🙋‍♂️ Hmm. . . . . . I forget. Help me remember!
- 🙋‍♂️ Yeah, I think they’re callback functions.

Continuations?

Do you *remember* continuations?

- 🙋‍♂️ Hmm. . . . . . I forget. Help me remember!
- 🙋‍♂️ Yeah, I think they're callback functions.
- 👴 I'm one with continuations.

●Continuations?

Do you *remember* continuations?

- 🙋‍♂️ Hmm. . . . . . I forget. Help me remember!
- 🙋‍♂️ ✅ **Yeah, I think they're callback functions.**
- 👴 I’m one with continuations.

Yes, they are just
callback functions!

●Continuations?

E.g. Read a file, then pass the result to a callback

```
readFile(file).flatMap { data ⇒ ...... }
```

●Continuations?

E.g. Read a file, then pass the result to a callback

```
readFile(file).flatMap { data => ...... }
```

continuation

Continuations?

E.g. Read a file, then pass the result to a callback

```
readFile(file).flatMap { data ⇒ ...... }
```

Continuations?

E.g. Read a file, then pass the result to a callback

```
readFile(file).flatMap { data ⇒ ...... }
```

```
for {
    data ← readFile(file)
    ......
} yield res
```

with for

Continuations?

E.g. Read a file, then pass the result to a callback

```
readFile(file).flatMap { data ⇒ ...... }
```

desugar

with for

```
for {
  data ← readFile(file)
  ......
} yield res
```

●Continuations?

Just a callback? ??

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

Continuations?

Just a callback? ??

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
           f()     ▷
((v1) ⇒ g(v1)  ▷
((v2) ⇒ h(v2)  ▷
((v3) ⇒   ......  )))
```

```
def x ▷ k ≡ k(x)
```

Pseudo Continuation-Passing Style Conversion

!?

Continuations?

Just a callback? ??

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
              f()     ▷
((v1) ⇒ g(v1) ▷
((v2) ⇒ h(v2) ▷
((v3) ⇒ ......    )))
```

```
def x ▷ k ≡ k(x)
```

Pseudo Continuation-Passing Style Conversion

!?

●Continuations?

Just a callback? ??

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
         f()      ▷
((v1) ⇒ g(v1) ▷
((v2) ⇒ h(v2) ▷
((v3) ⇒ ......   )))
```

def x ▷ k ≡ k(x)

Pseudo Continuation-Passing Style Conversion

!?

Continuations?

Just a callback? ??

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
       f()      ▷
((v1) ⇒ g(v1)   ▷
((v2) ⇒ h(v2)   ▷
((v3) ⇒ ......   )))
```

```
def x ▷ k == k(x)
```

Pseudo Continuation-Passing Style Conversion

⁉

●Continuations?

Just a callback? **No —**

Continuations are . . . . . .

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
          f()     ▷
((v1) ⇒  g(v1)   ▷
((v2) ⇒  h(v2)   ▷
((v3) ⇒   ......  )))
```

```
def x ▷ k ≡ k(x)
```

Pseudo Continuation-Passing Style Conversion

⁉

Continuations?

Just a callback? **No —**

Continuations are . . . . . .

The Rest of The Computation!!

```
val v1 = f()
val v2 = g(v1)
val v3 = h(v2)
......
```

```
          f()    ▷
((v1) ⇒  g(v1)  ▷
((v2) ⇒  h(v2)  ▷
((v3) ⇒   ......  )))
```

```
def x ▷ k ≡ k(x)
```

Pseudo Continuation-Passing Style Conversion

!?

Continuations:
continued and to be continued

Dump and J operator, SECD machine

The first abstract machine for evaluating functional programs[4]:

`Stack`	:	Stores intermediate results
`Environment`	:	Stores variables
`Control`	:	Stores the next instruction
`Dump`	:	Stores the suspended computation

Dump and J operator, SECD machine

The first abstract machine for evaluating functional programs[4]:

Stack	:	Stores intermediate results
Environment	:	Stores variables
Control	:	Stores the next instruction
Dump	:	Stores the suspended computation

And J operator captures the Dump[10]

Dump and J operator, SECD machine

The first abstract machine for evaluating functional programs[4]:

Stack	:	Stores intermediate results
Environment	:	Stores variables
Control	:	Stores the next instruction
Dump	:	Stores **the suspended computation**

And J operator **captures the** Dump[10]

This marks the origin of **continuations!**[17][7]

call/cc: goto Practical

Operators in Scheme[16], notably `call/cc`[15][*2]:
Captures the current continuation as a first-class value

```
(let {[(x (call/cc (λ (K)
                     (+ 2 (K 4)))))]}
    (+ x 3))
; ⇒ (let {[x 4]} (+ x 3))
; ⇒ returns 7
```

[*2] Short of ”call-with-current-continuation”

●call/cc: goto Practical

Operators in Scheme[16], notably `call/cc`[15][*2]:
Captures the current continuation as a first-class value

```
(let {[(x (call/cc (λ (K)
                     (+ 2 (K 4)))))]}
     (+ x 3))
; ⇒ (let {[x 4]} (+ x 3))
; ⇒ returns 7
```

`call/cc` can be used to implement:

- ✅ non-local exit
- ✅ backtracking

  ……

[*2] Short of ”call-with-current-continuation”

call/cc: goto Practical

Operators in Scheme[16], notably `call/cc`[15][*2]:
Captures the current continuation as a first-class value

```
(let {[(x (call/cc (λ (K)
                     (+ 2 (K 4)))))]}
    (+ x 3))
; ⇒ (let {[x 4]} (+ x 3))
; ⇒ returns 7
```

`call/cc` can be used to implement:

- ✅ non-local exit
- ✅ backtracking
- ......

👉 **ANYTHING related to *control transfer***

[*2] Short of "call-with-current-continuation"

Delimited Continuations

Problem: `call/cc` is a powerful control structure, but captures the entire rest of the computation (like `goto`!)

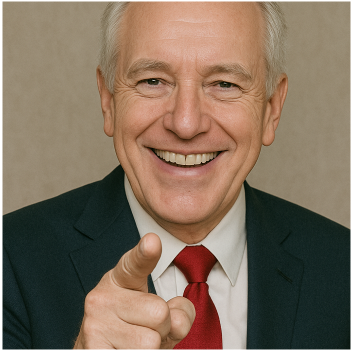
Delimited Continuations

Problem: `call/cc` is a powerful control structure, but captures the entire rest of the computation (like `goto`!)

Solution:

Use
Delimited Continuations! [9]

- ✅ capturing ~~*scoped*~~ delimited continuations
- ✅ more structured and composable



Delimited Continuations

```
(+ 3 (reset (+ 4 (shift κ (κ (κ 5))))))

; ⇒ (+ 3 (+ 4 (+ 4 5)))
```

Delimited Continuations

```
(+ 3 (reset (+ 4 (shift κ (κ (κ 5))))))

; ⇒ (+ 3 (+ 4 (+ 4 5)))
```

Several variants of operators:

- control/prompt[9] $\texttt{control}_0/\texttt{prompt}_0$[13]
- shift/reset[6] $\texttt{shift}_0/\texttt{reset}_0$[13]
- fcontrol/run[14]
- *multiprompt* extensions[8]
- ……

Delimited Continuations

```
(+ 3 (reset (+ 4 (shift κ (κ (κ 5))))))

; ⇒ (+ 3 (+ 4 (+ 4 5)))
```

Several variants of operators: 

- control/prompt[9] $\texttt{control}_0/\texttt{prompt}_0$[13]
- shift/reset[6] $\texttt{shift}_0/\texttt{reset}_0$[13]
- fcontrol/run[14]
- *multiprompt* extensions[8]
- ······

Delimited continuations enable us to implement

✅ `call/cc`!

✅ **ALL monads!!**

Delimited Continuations

```
(+ 3 (reset (+ 4 (shift κ (κ (κ 5))))))

; ⇒ (+ 3 (+ 4 (+ 4 5)))
```

Several variants of operators: 

- control/prompt[9] $\texttt{control}_0$/$\texttt{prompt}_0$[13]
- shift/reset[6] $\texttt{shift}_0$/$\texttt{reset}_0$[13]
- fcontrol/run[14]
- *multiprompt* extensions[8]
- ……

Delimited continuations enable us to implement

✅ call/cc!

✅ **ALL monads!!**

…… and **vice-versa!**[8]

call/cc

delimited controls ⟷ monads

Continuation-Passing Style

A program representation where control flow is made *explicit* by chaining computations as continuations[12]:

```
(define (add1 x)
  (+ x 1))
(define (mul2 x)
  (* x 2))

(mul2 (add1 3))
```

Continuation-Passing Style

A program representation where control flow is made *explicit* by chaining computations as continuations[12]:

```
(define (add1 x)
  (+ x 1))
(define (mul2 x)
  (* x 2))

(mul2 (add1 3))
```

CPS Conversion!

```
(define (add1 x κ)
  (κ (+ x 1)))
(define (mul2 x κ)
  (κ (* x 2)))

(add1 3 (λ (smu)
  (mul2 sum (λ (mul)
    mul))))
```

Continuation-Passing Style

A program representation where control flow is made *explicit* by chaining computations as continuations[12]:

```
(define (add1 x)
  (+ x 1))
(define (mul2 x)
  (* x 2))

(mul2 (add1 3))
```

CPS Conversion!

```
(define (add1 x κ)
  (κ (+ x 1)))
(define (mul2 x κ)
  (κ (* x 2)))

(add1 3 (λ (smu)
  (mul2 sum (λ (mul)
    mul))))
```

CPS *fixes* the order of evaluation and control flow

Continuation-Passing Style

A program representation where control flow is made *explicit* by chaining computations as continuations[12]:

```
(define (add1 x)
  (+ x 1))
(define (mul2 x)
  (* x 2))
                ①②
(mul2 (add1 3))
```

CPS Conversion! →

```
(define (add1 x κ)
  (κ (+ x 1)))
(define (mul2 x κ)
  (κ (* x 2)))
      ①
(add1 3 (λ (smu)
  (mul2 sum (λ (mul)
    mul))))②
```

CPS *fixes* the order of evaluation and control flow, so that it's a good choice for

an intermediate representation

for language implementations!

Compiling with Continuations

CPS as an intermediate representation (IR) for language impls[1]:

$$\boxed{\text{Source}} \longrightarrow \boxed{\text{CPS}} \xrightarrow[\text{(Opts.)}]{} \boxed{\text{Codegen}}$$

●Compiling with Continuations

CPS as an intermediate representation (IR) for language impls[1]:

SML/NJ
Scheme
OCaml
(flambda wasm)

Source → CPS → (Opts.) → Codegen

✅ Good for **functional languages**

CPS operates functions as first-class values!

Compiling with Continuations

CPS as an intermediate representation (IR) for language impls[1]:

SML/NJ
Scheme
OCaml
(flambda
wasm)

Source → CPS → (Opts.) → Codegen

Constant Folding & Inlining	⇒ β reduction
Defunctionalization	⇒ η reduction
Common Subexpression Elimination	⇒ *EZ*:

```
let z = a * b + a * b
in e
```

same!

CPS Conversion ($[\![-]\!]$)

```
( *' ) a b (fun v₁ ->
( *' ) a b (fun v₂ ->
( +' ) v₁ v₂ (fun z ->
⟦e⟧
```

✅ Good for **functional languages**

CPS operates functions as first-class values!

✅ Good for **optimizations**

Several optimizations can be done by β/η,
and each values are single-assignment!

Correspondence between procedural

How do *control transfers* correspond in functional and procedural?

- Functional side:

- Procedural side:

---

*3 Static Single-Assignment form

●Correspondence between procedural

How do *control transfers* correspond in functional and procedural?

- Functional side:
  - ⬇️ J: Low-level continuation operator
- Procedural side:
  - ⬇️ `jmp`: Low-level control transfer

---

*3 Static Single-Assignment form

●Correspondence between procedural

How do *control transfers* correspond in
functional and procedural?

- Functional side:
  - ⬇️ `J`: Low-level continuation operator
  - ⬇️ `call/cc`: Capturing entire continuation
- Procedural side:
  - ⬇️ `jmp`: Low-level control transfer
  - ⬇️ `goto`: Arbitrary jumps in high-level repr.

[*3] Static Single-Assignment form

●Correspondence between procedural

How do *control transfers* correspond in
functional and procedural?

- ▶ Functional side:
  - ⬇️ J: Low-level continuation operator
  - ⬇️ `call/cc`: Capturing entire continuation
  - 👉 `shift/reset`: Structured, modular and composable control
- ▶ Procedural side:
  - ⬇️ `jmp`: Low-level control transfer
  - ⬇️ `goto`: Arbitrary jumps in high-level repr.
  - 👉 `for` / `while` / `if`: Structured, clear control[5]

*3 Static Single-Assignment form

Correspondence between procedural

How do *control transfers* correspond in
functional and procedural?

- Functional side:
  - ⬇️ J: Low-level continuation operator
  - ⬇️ `call/cc`: Capturing entire continuation
  - 👉 `shift/reset`: Structured, modular and composable control
  - ⚙️ **CPS as an IR**
- Procedural side:
  - ⬇️ `jmp`: Low-level control transfer
  - ⬇️ `goto`: Arbitrary jumps in high-level repr.
  - 👉 `for` / `while` / `if`: Structured, clear control[5]
  - ⚙️ **SSA**[*3] **as an IR**

And also,
CPS $\Leftrightarrow$ **SSA!** [2]

[*3] Static Single-Assignment form

Continuations:

continued and to be continued

Why continuations for effects?

Effects, side effects or computational effects:

- Exception
- Async I/O
- Coroutines
- etc.

Handling effects is about what happens——

Why continuations for effects?

Effects, side effects or computational effects:

- Exception . . . . *halt?*
- Async I/O . . . . *when to resume?*
- Coroutines . . . . *which order to resume?*
- etc.

Handling effects is about what happens——

and *when* and *how* to **resume**.

●Why continuations for effects?

Effects, side effects or computational effects:

- Exception .... *halt?*
- Async I/O ....*when to resume?*
- Coroutines .... *which order to resume?*
- etc.

Handling effects is about what happens——
and *when* and *how* to resume.

So, it's time for

Continuations!

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

continuation

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a        continuation
  (>>=)  :: m a -> (a -> m b) -> m b
```

```
instance Monad Maybe where
  return = Just
  Just x  >>= k = k x
  Nothing >>= _ = Nothing
```

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a      continuation
  (>>=)  :: m a -> (a -> m b) -> m b
```

```
maybe e >>= \x -> u
```

```
instance Monad Maybe where
  return = Just
  Just x  >>= k = k x
  Nothing >>= _ = Nothing
```

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a        continuation
  (>>=)  :: m a -> (a -> m b) -> m b
```

```
instance Monad Maybe where
  return = Just
  Just x  >>= k = k x
  Nothing >>= _ = Nothing
```

```
maybe e >>= \x -> u
```

⇓ direct-style with do

```
do
  x <- maybe e
  u
```

Monads

An approach to modeling calculi with *effects*[11]:

```
class Monad m where
  return :: a -> m a        continuation
  (>>=)  :: m a -> (a -> m b) -> m b
```

```
instance Monad Maybe where
  return = Just
  Just x  >>= k = k x
  Nothing >>= _ = Nothing
```

throw away continuation
to stop computation!

```
maybe e >>= \x -> u
```

⇓ direct-style with do

```
do
  x <- maybe e
  u
```

Algebraic Effect Handlers

A new way to model effectful computations, with high *modularity* and *composablility*[3]:

```
type _ eff += Print : string -> unit eff

match perform @@ Print "hello" with
| effect (Print msg),  k ->
  print_endline msg;  continue k ()
```

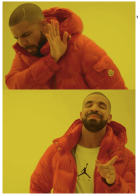
Algebraic Effect Handlers

A new way to model effectful computations, with high *modularity* and *composablility*[3]:

```
type _ eff += Print : string -> unit eff

match perform @@ Print "hello" with
| effect (Print msg), k ->
  print_endline msg; continue k ()
```

Handlers make it *easier* to **compose** and **modularise** than monad transformers!


[*a]

Monad Transformers

Algebraic Effect Handlers

[*a] https://www.youtube.com/watch?v=uxpDa-c-4Mc

Conclusion

Summary

- Explain from the history to recent trends of **continuations**
- Continuations are powerful, joyful, and useful!
- Research about continuations is **still hot topic**, and **continued!**

Couldn't talk today 😭

- Type Systems
  - Answer-Type Modification
  - Linear Types for efficient runtime repr
  - Type-preserving conversion
- Formal Semantics for Natural Languages
- Recursion
- ······ and *anything* about computation with continuations.

References I

[1] Andrew W. Appel. *Compiling with continuations*. USA: Cambridge University Press, 1992. ISBN: 0521416957.

[2] Andrew W. Appel. "SSA is functional programming". In: *SIGPLAN Not.* 33.4 (Apr. 1998), pp. 17–20. ISSN: 0362-1340. DOI: 10.1145/278283.278285. URL: https://doi.org/10.1145/278283.278285.

[3] Andrej Bauer and Matija Pretnar. "An Effect System for Algebraic Effects and Handlers". In: vol. 10. June 2013. ISBN: 978-3-642-40205-0. DOI: 10.1007/978-3-642-40206-7_1.

[4] W. H. Burge. "The evaluation, classification and interpretation of expressions". In: *Proceedings of the 1964 19th ACM National Conference*. ACM '64. New York, NY, USA: Association for Computing Machinery, 1964, pp. 11.401–11.4022. ISBN: 9781450379182. DOI: 10.1145/800257.808888. URL: https://doi.org/10.1145/800257.808888.

[5] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured programming*. GBR: Academic Press Ltd., 1972. ISBN: 0122005503.

[6] Olivier Danvy and Andrzej Filinski. “Abstracting control”. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP ’90. Nice, France: Association for Computing Machinery, 1990, pp. 151–160. ISBN: 089791368X. DOI: `10.1145/91556.91622`. URL: `https://doi.org/10.1145/91556.91622`.

[7] Olivier Danvy and Kevin Millikin. “A Rational Deconstruction of Landin’s J Operator”. In: *Implementation and Application of Functional Languages*. Ed. by Andrew Butterfield, Clemens Grelck, and Frank Huch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 55–73. ISBN: 978-3-540-69175-4.

[8] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. “A monadic framework for delimited continuations”. In: *J. Funct. Program.* 17.6 (Nov. 2007), pp. 687–730. ISSN: 0956-7968. DOI: `10.1017/S0956796807006259`. URL: `https://doi.org/10.1017/S0956796807006259`.

[9] Mattias Felleisen. “The theory and practice of first-class prompts”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 180–190. ISBN: 0897912527. DOI: `10.1145/73560.73576`. URL: `https://doi.org/10.1145/73560.73576`.

[10] Peter J. Landin. “A Generalization of Jumps and Labels”. In: *Higher Order Symbol. Comput.* 11.2 (Sept. 1998), pp. 125–143. ISSN: 1388-3690. DOI: `10.1023/A:1010068630801`. URL: `https://doi.org/10.1023/A:1010068630801`.

[11] Eugenio Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(91)90052-4`. URL: `https://www.sciencedirect.com/science/article/pii/0890540191900524`.

[12] John C. Reynolds. “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM ’72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

[13] Chung-chieh Shan. “Shift to control”. In: *Proceedings of the 5th workshop on Scheme and Functional Programming*. 2004, pp. 99–107.

[14] Dorai Sitaram. “Handling control”. In: *SIGPLAN Not.* 28.6 (June 1993), pp. 147–155. ISSN: 0362-1340. DOI: 10.1145/173262.155104. URL: https://doi.org/10.1145/173262.155104.

[15] Guy L Steele Jr and Gerald Jay Sussman. *The Revised Report on SCHEME: A Dialect of LISP*. Tech. rep. AI Memo 452. MIT Artificial Intelligence Laboratory, 1978. URL: https://dspace.mit.edu/handle/1721.1/6283.

[16] Gerald Jay Sussman and Guy L. Steele Jr. *SCHEME: An Interpreter for Extended Lambda Calculus*. Tech. rep. AI Memo 349. MIT Artificial Intelligence Laboratory, 1975. URL: https://dspace.mit.edu/handle/1721.1/5794.

[17] Hayo Thielecke. "An Introduction to Landin 's "A Generalization of Jumps and Labels"". In: *Higher Order Symbol. Comput.* 11.2 (Sept. 1998), pp. 117–123. ISSN: 1388-3690. DOI: 10.1023/A:1010060315625. URL: https://doi.org/10.1023/A:1010060315625.