Understanding Continuations

Frank Terbeck
ft@bewatermyfriend.org

October 2, 2019

Continuations?

Simple Example

```
(+ 5 4)
```

Simple Example

```
(+ 5 4) → 9
```

Simple Example

```
(lambda (x) (* x x))
```

Simple Example

```
(define (square x) (* x x))
```

Simple Example

```
                (square 5) → 25
((lambda (x) (* x x)) 5) → 25
```

Simple Example

```
(if Condition
  Consequence
  Alternative)

(if (zero? 3)
  (display "foo")
  (display "bar"))
```

Main Strategy

- Computation: `(+ 5 4)`
- Task: Express Continuation at the point of `5`

- Idea: Take perspective of the `5`
- With that, find a function:
  - …that takes one argument…
  - …and represents the rest of the computation
  - …after `5` is evaluated.
- *That* is our continuation!

Our First Continuation

idea: `(+ _ 4)`

```
(define (kont v)
  (+ v 4))

(kont  5) → 9
(kont 10) → 14
```

Beyond Theory — `call/cc`

Beyond Theory

- Thus far continuations are thought experiments.
- Scheme: `call-with-current-continuation`
- …shorthand: `call/cc`
- Its jobs:
  - Capture a continuation.
  - Make it accessible to the programmer.

Beyond Theory

- Signature: `(call/cc fnc)`
- `fnc` takes one argument: The continuation `k`.
- Return value of `call/cc`: The value of `(fnc k)`.
- Example: `(call/cc (lambda (k) 5))` → `5`

Storing Continuations

```
(+ 5 4) → 9

(define kont #f)
(define (five k)
  (set! kont k)
  5)

(+ (call/cc five) 4) → 9
```

Storing Continuations

```
kont → #<continuation>

(kont  5) →  9
(kont 10) → 14
```

Context is important

- `kont` looks and feels like a function.
- But `kont → #<continuation>` — *not* `→ #<procedure>`
- The captured continuation is *not* just a function.
- When called upon, it invokes the entire original context!

Practicality

- Product: `(fold * (*) '(1 2 3 4 0 6 7 8 9 10))` → `0`
- How about early exit:

```
(call/cc (lambda (return)
           (fold (lambda (element accumulator)
                   (if (zero? element)
                     (return 0)
                     (* element accumulator)))
                 (*)
                 '(1 2 3 4 0 6 7 8 9 10))))  → 0
```

Conclusion

- Continuations are a powerful control flow primitive.
- Enables implementation of things like:
  - Early Exit
  - Exceptions
  - Co-Routines
  - Generators
  - …and more.
- Facilities can live in *libraries* rather than a language standard.

- Blog: `http://bewatermyfriend.org/p/2019/002/`
- Source: `https://github.com/ft/continuations`

Fin?

Mind Bending

Mind Bending

- `(((call/cc identity) identity) "Hey!")` → `"Hey!"`
- Confusion Density Maximum[1]
- identity: `(lambda (x) x)`

[1] According to R. Kent Dybvig.

Mind Bending

- `(((call/cc identity) identity) "Hey!")` → `"Hey!"`
- Since: `(operator operand)` → `operand`
- `operator` has to be `identity`[2]

[2] Barring any side-effects.

Mind Bending

- `((call/cc identity) identity)` → `identity`
- Use strategy with: `(_ identity)`
- What is a function that:
  - …represents the rest of the computation…
  - …from the perspective of `_`?
- Answer:

```
(lambda (v)
  (v identity))
```

Mind Bending

```
To show:   ((call/cc identity) identity) → identity

continuation:            (lambda (v) (v identity))
call/cc:       (identity (lambda (v) (v identity)))
into expr:              ((lambda (v) (v identity)) identity)
                           → (identity identity)
                           → identity

          (((call/cc identity) identity) "Hey!")
          (identity                      "Hey!") → "Hey!"
```

Thanks for your attention!