~~Kontinyueshon~~

Continuation

the Ultimate GOTO

suhorng

GO TO

- A Brief, Incomplete, and Mostly Wrong History of Programming Languages

1964 - John Kemeny and Thomas Kurtz create BASIC, an unstructured programming language for non-computer scientists.

1965 - Kemeny and Kurtz go to 1964.

A "Goto" with Context

- Jump!

```
void dfs(int u) {
  for (int i = 0; i != deg[u]; ++i) {
    int v = edges[u][i];
    if (!vst[v]) {
      vst[v] = true;
      dfs(v);
    }
  }
}
```

- Make it a loop!

A "Goto" with Context

- `GOTO` is insufficient

```
struct frame_t { int u, i, addr; };
void dfs2(int u) {
  vector<frame_t> stk;
  stk.push_back({u, 0, -1});
call:
  for (; stk.back().i != deg[stk.back().u]; ++stk.back().i) {
    int v; v = edges[stk.back().u][stk.back().i];
    if (!vst[v]) {
      vst[v] = true;
      stk.push_back({v, 0, 1});
      goto call;
ret1:;
    }
  }
ret2:
  switch (stk.back().addr) {
    case 1: stk.pop_back(); goto ret1;
  }
}
```

A "Goto" with Context

- Time for the Continuation!
  - The rest of the computation

$\texttt{let}\ x = 5 + 6 \times 3\ \texttt{in}\ f\ (f\ x)$

A "Goto" with Context

- Time for the Continuation!
    - The rest of the computation

$\texttt{let}\ x = 5 + 6 \times 3\ \texttt{in}\ f\ (f\ x)$

$\leadsto \texttt{let}\ x = 5 + 18\ \texttt{in}\ f\ (f\ x)$

A "Goto" with Context

- Time for the Continuation!
  - The rest of the computation

$\texttt{\textbf{let}}\ x = 5 + 6 \times 3\ \texttt{\textbf{in}}\ f\ (f\ x)$

$\rightsquigarrow \texttt{\textbf{let}}\ x = 5 + 18\ \texttt{\textbf{in}}\ f\ (f\ x)$

$\rightsquigarrow \texttt{\textbf{let}}\ x = 23\ \texttt{\textbf{in}}\ f\ (f\ x)$

A "Goto" with Context

- Time for the Continuation!
    - The rest of the computation

$$\texttt{let}\ x = 5 + 6 \times 3\ \texttt{in}\ f\ (f\ x)$$

$$\leadsto \texttt{let}\ x = 5 + 18\ \texttt{in}\ f\ (f\ x)$$

$$\leadsto \texttt{let}\ x = 23\ \texttt{in}\ f\ (f\ x)$$

$$\leadsto f\ (f\ 23)$$

A "Goto" with Context

- Time for the Continuation!
    - The rest of the computation

$$\texttt{let}\ x = 5 + 6 \times 3\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow \texttt{let}\ x = 5 + 18\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow \texttt{let}\ x = 23\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow f\ (f\ 23)$$

$$\rightsquigarrow f\ (???_1)$$

A "Goto" with Context

- Time for the Continuation!
  - The rest of the computation

$$\texttt{let}\ x = 5 + 6 \times 3\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow \texttt{let}\ x = 5 + 18\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow \texttt{let}\ x = 23\ \texttt{in}\ f\ (f\ x)$$

$$\rightsquigarrow f\ (f\ 23)$$

$$\rightsquigarrow f\ (???_1)$$

$$\rightsquigarrow ???_2$$

A "Goto" with Context

- Time for the Continuation!
  - The rest of the computation

“$\texttt{let}\ x = 5 + [\,]\ \texttt{in}\ f\ (f\ x)$” and “$6 \times 3$”

“$\texttt{let}\ x = [\,]\ \texttt{in}\ f\ (f\ x)$” and “$5 + 18$”

“” and “$\texttt{let}\ x = 23\ \texttt{in}\ f\ (f\ x)$”

“$f\ ([\,])$” and “$f\ 23$”

“” and “$f\ ???_1$”

A "Goto" with Context

$$``E[R] := 1 + 2 \times (f\ x)\ "$$

- The program can be decomposed into two parts
  - Current computation, or a “redex”
    - $R := (f\ x)$
  - Rest of the computation, the “evalutaion context”
    - $E[] := 1 + 2 \times\ \ [\ ]$

Representation of the continuation

- As a "function" (not quite)
- The famous `call/cc` operator

```
(define fast-product
  (lambda (xs)
    (call/cc            ; call/cc is a built-in operator
     (lambda (k)
       (letrec
           ([prod
             (lambda (xs)
               (cond
                [(null? xs) 1]
                [(zero? (car xs)) (k 0)]
                [else (* (car xs) (prod (cdr xs)))]))])
         (prod xs))))))
```

`call/cc` in Scheme

- What it does
  - Capture the whole $E[\,]$ evaluation context and make it a first-class "value"
  - When the (captured) continuation is applied, discard the current evalution context and installs the captured one
- Define the abort operator $\texttt{abort}(M)$ to be

  $E[\texttt{abort}(M)] \rightsquigarrow M$, i.e. abandoning $E[\,]$
- `call/cc` is then

  $E[\texttt{call/cc}(f)] \rightsquigarrow E[f\,(\,\lambda\texttt{v}.\ \texttt{abort}(E[\texttt{v}])\,)]$

`call/cc` in Scheme

$$1 + 5 \times (\texttt{call/cc}(\lambda\texttt{k}.\ 3 - \texttt{k}(2))) \qquad (= E_1[R_1])$$

$$R_1 = \texttt{call/cc}(\lambda\texttt{k}.\ 3 - \texttt{k}(2))$$

$$E_1[\,] = 1 + 5 \times [\,]$$

`call/cc` in Scheme

$$1 + 5 \times (\texttt{call/cc}(\lambda\texttt{k}.\ 3 - \texttt{k}(2))) \qquad (= E_1[R_1])$$

$$R_1 = \texttt{call/cc}(\lambda\texttt{k}.\ 3 - \texttt{k}(2))$$

$$E_1[\,] = 1 + 5 \times [\,]$$

$$\rightsquigarrow 1 + 5 \times (3 - \texttt{k}(2)) \qquad (= E_2[R_2])$$

$$R_2 = \texttt{k}(2)$$

$$E_2[\,] = 1 + 5 \times (3 - [\,])$$

$$\texttt{k} = E_1[\,] = 1 + 5 \times [\,]$$

`call/cc` in Scheme

$$1 + 5 \times (\mathtt{call/cc}(\lambda\mathtt{k}.\ 3 - \mathtt{k}(2))) \qquad (= E_1[R_1])$$

$$R_1 = \mathtt{call/cc}(\lambda\mathtt{k}.\ 3 - \mathtt{k}(2))$$

$$E_1[\,] = 1 + 5 \times [\,]$$

$$\rightsquigarrow 1 + 5 \times (3 - \mathtt{k}(2)) \qquad (= E_2[R_2])$$

$$R_2 = \mathtt{k}(2)$$

$$E_2[\,] = 1 + 5 \times (3 - [\,])$$

$$\mathtt{k} = E_1[\,] = 1 + 5 \times [\,]$$

$\rightsquigarrow 1 + 5 \times 2$ $\qquad (= E_3[R_3])$; $E_1[\,]$ is installed with $[\,] = 2$

$$R_3 = 5 \times 2$$

$$E_3[\,] = 1 + [\,]$$

`call/cc` in Scheme

$$1 + 5 \times (\texttt{call/cc}(\lambda \texttt{k}.\ 3 - \texttt{k}(2))) \qquad (= E_1[R_1])$$

$$R_1 = \texttt{call/cc}(\lambda \texttt{k}.\ 3 - \texttt{k}(2))$$

$$E_1[\,] = 1 + 5 \times [\,]$$

$$\rightsquigarrow 1 + 5 \times (3 - \texttt{k}(2)) \qquad (= E_2[R_2])$$

$$R_2 = \texttt{k}(2)$$

$$E_2[\,] = 1 + 5 \times (3 - [\,])$$

$$\texttt{k} = E_1[\,] = 1 + 5 \times [\,]$$

$\rightsquigarrow 1 + 5 \times 2$ $\qquad (= E_3[R_3])$; $E_1[\,]$ is installed with $[\,] = 2$

$$R_3 = 5 \times 2$$

$$E_3[\,] = 1 + [\,]$$

$$\rightsquigarrow 1 + 10$$

$$\rightsquigarrow 11$$

In Meta-language: Continuation-Passing Style (CPS)

- A computation of type `a`:
  - `(a → w) → w`
- Continuation expecting a value of type `a`:
  - A function `a → w`
- A function of type `a → b`:
  - A transformation `(b → w) → (a → w)`,
    or equivalently `a → (b → w) → w`

In Meta-language: Continuation-Passing Style (CPS)

```
(lambda (f)              ; direct style
  (lambda (x)
    (f (f x))))
```

```
(lambda (f k0)            ; in CPS
  (k0 (lambda (x k1)
        (f x (lambda (v)
               (f v k1))))))
```

```
function(f, k0) {         // in javascript...
  return k0(function(x, k1) {
    return f(x, function(v) {
      return f(v, k1);
    })
  });
}
```

Implementing `call/cc` for CPS programs

```
(define call/cc          ; the implementatin of call/cc in CPS
  (lambda (f k)
    (f (lambda (v k1) ; (i)   capture the continuation `k`
         (k v))          ; (ii)  when applied, abandon `k1`
       k)))              ; (iii) `k` for normal return
```

- An example: $1 + 5 \times \mathtt{call/cc}(\lambda \mathtt{k}.\ 3 - \mathtt{k}(2))$

```
(call/cc                 ; in CPS;
  (lambda (k k2)         ; not the original `call/cc` operator
     (k 2 (lambda (v)
            (k2 (- 3 v)))))
  (lambda (v2)
     (+ 1 (* 5 v2))))
```

CPS Transformation

- Value:
  - $\mathcal{T}(\mathtt{x}) = \lambda\mathtt{k}.\ \mathtt{k}\ \mathtt{x}$
- Function:
  - $\mathcal{T}(\lambda\mathtt{x}.\,\mathtt{e}) = \lambda\mathtt{x}.\,\lambda\mathtt{k}.\ \mathcal{T}(\mathtt{e})\ \mathtt{k}$
- Application:
  - $\mathcal{T}(\mathtt{e}_1\ \mathtt{e}_2) = \mathcal{T}(\mathtt{e}_1)\ \ (\lambda\mathtt{v}_1.\ \mathcal{T}(\mathtt{e}_2)\ (\lambda\mathtt{v}_2.\ \mathtt{v}_1\ \mathtt{v}_2\ \mathtt{k}))$

Static Single Assignment Form (SSA)

- Every variable can be assigned once -- one can view it as definition

```
z  ←  x + y
b  ←  z * 3

if (b > 0)
    u  ←  5
else
    v  ←  2

w  ←  φ(u, v)
...
```

- There is a special φ function, combining values from different control paths

CPS v.s. SSA

```
z  ←  x + y
b  ←  z * 3
if (b > 0)
    u  ←  5
else
    v  ←  2
w  ←  φ(u, v)
...
```

```
(let ([k (lambda (w)
            ...)])

  (+ x y (lambda (z)
    (* z 3 (lambda (b)
      (if (> b 0)
        (k 5)
        (k 2))))))))
```

Others

- CPS/ANF: An intermediate representation in the compiler

```
(let ([k (lambda (w)
           ...)])
(let ([z (+ x y)])     ; There are no expressions like
(let ([b (* z 3)]))    ; (let ([b (* (+ x y) 3)]) ...)
(if (> b 0)
  (k 5)
  (k 2))))
```

- Delimited continuation (better than undelimited one)

```
(define (shift f k)   ; (its implementation in CPS)
  (f (lambda (v k1) (k1 (k v))) ; compose the cont.
     id))

(define (reset e k)   ; (its implementation in CPS)
  (k (e id)))
```

Others

- continuation ⇔ classical logic
- (delimited) continuation ⇒ generator, coroutine, exception
  - They are specialized operations on the continuation
- CPS: Simple syntax ⇒ a good intermediate form
  - ~~I shall return~~There is no “return”; all are tail calls

```
v   ::=   x   |   (λ x. e)

e   ::=   (v1 v2 ...)
```

- Interpreter ⇔ CPS ⇔ Stack Machine