Continuations and web servers
AMP, Spring 2009

Morten Rhiger (mir@itu.dk)
March 18, 2009
Today’s plan

- Brief HTTP overview
- Web servers and continuations
- First-class continuations and call/cc
- A WebServer-Session Simulator in Scheme
- Client (browser) submit requests; server responds, producing HTML
- With forms, information is also send from client to server.
- How do servers serve many client simultaneously? How do servers deal with clients that “go back” (by hitting the browsers Back button)?
Example HTML with forms

```html
<html>
  ...
  <form method="POST" action="http://www.myserver.com">
    ...
    <input type="TEXT" name="number"/>
    <input type="SUBMIT" value="Send"/>
    ...
  </form>
</html>
```

- Resulting HTML has one text field (`number`) and a button (titled `Send`).
- When hitting `Send`, the server receives an HTTP request containing
  - The URL of the form (here `http://www.myserver.com`)
  - Values for all form fields (here `number`)
Client–server communication

- A client–server session may require several requests and responses.

- (Note that, seen from the servers perspective, a response is a question asked to the client whereas a request is an answer received from the client.)

- Unlike traditional desktop GUI applications:
  - A server cannot update the client. (It can only respond to requests.)
  - A client may go back in history, sending a request again (possible with different form-field values),
Server-driven communication

- View an interaction as a question–answering session
  - Client initiates the session
  - Server asks questions by responding to requests
  - Client answers to questions by sending requests (containing form-field values)

- When server asks a questions, it remembers its state and stops running.

- Once the client answers to a question, the remembered state is reinstalled and the server continues executing where it left.
The current state of the server = the continuation!

Example server:

```c
int n1 = input_from_client("Enter a number");
int n2 = input_from_client("Ok, got " + n1 + ". Now enter another");
return n1 + n2;
```

The first `input_from_client` makes the server stop after having send the proper response to the client.

What is the continuation at this point? How do we get it? How do we store it?
First-class continuations

- From implicit continuations ("under the hood") to explicit first-class continuations
- In Scheme:
  
  ```scheme
  ... (call/cc (lambda (k) ... (k v) ...))
  ```
  
  k is a function representing the continuations of `(call/cc ...)`. It can be treated as any other function. (E.g., it is first class)

- In Standard ML of New Jersey (in module SMLofNJ.Cont):
  
  ```haskell
  ... (callcc (fn (k) => ... throw k v ...))
  ```
  
  k is a representation of the continuations of `(callcc ...)`. Again, it is first class, but it is not a function: To “apply” it, we must use `throw`. 
Call/cc in Scheme and SML/NJ

- Scheme:
  > (+ 1 (call/cc (lambda (k) (+ 10 (k 100)))))
  101

- SML/NJ
  - 1 + (callcc (fn k => 10 + (throw k 100)));
  val it = 101 : int
datatype exp
   = INT of int
   | ADD of exp * exp
   | VAR of string
   | LET of string * exp * exp
   | CAPTURE of string * exp (* capture K in E *)
   | THROW of string * exp (* throw K E *)
Runtime values includes integers (as in the previous interpreters) but also continuations. (Captured continuations are stored in variables.)

```plaintext
datatype value
    = I of int
    | C of value -> value
```

Interpreter for call/cc

(Run-time values)
fun eval (INT i)      env k = k (I i)
| eval (VAR x)       env k = k (lookup env x)
| eval (ADD (e1, e2)) env k = eval e1 env (fn I v1 =>
|                                           eval e2 env (fn I v2 => k (I (v1 + v2))))
| eval (LET (x, e1, e2)) env k = eval e1 env (fn v => eval e2 (bind env x v) k)
| eval (CAPTURE (x, e)) env k = eval e (bind env x (C k)) k
| eval (THROW (x, e)) env k = case (lookup env x) of C k’ => eval e env k’

fun run e = eval e [] (fn v => v)
Interpreter for call/cc

(Example)

- Concrete syntax:
  \[(\text{capture esc in } 1 + (\text{throw esc } 100)) + 10\]

- Abstract syntax
  \[
  \text{ADD} (\text{CAPTURE} ("esc", \text{ADD} (\text{INT} 1, \\
  \text{THROW} ("esc", \text{INT} 100))), \\
  \text{INT} 10)
  \]

- Result: 110

- Corresponding program in SML of New Jersey:
  \[(\text{callcc} (\text{fn esc => 1 + throw esc } 100)) + 10\]

- Corresponding program in Scheme:
  \[(+ (\text{callcc} (\lambda (esc) (+ 1 (esc 100)))))) 10\]
Interpreter for call/cc
(Trace of an example)

(capture esc in 1 + (throw esc 100)) + 10
  where k = fn v => v
-> 1 + (throw esc 100)
  where k = fn v => v
    esc = fn v => v + 10
-> throw esc 100
  where k = fn v => 1 + v
    esc = fn v => v + 10
-> esc 100
  where k = fn v => 1 + v
    esc = fn v => v + 10
-> 100 + 10
Expected “flow of control” in

```cpp
int n1 = input_from_client("Enter a number");
int n2 = input_from_client("Ok, got " + n1 + ". Now enter another");
return n1 + n2;
```

The server

1. grabs the `continuation` of the first line (the first call to `input_from_client` to be precise),
2. stores it under a identifier $K_1$,
3. constructs a page (saying “Enter a number”) that contains the identifier $K_1$,
4. sends the page to the client, and
5. stops running.
Web servers using continuations, continued 2

Expected “flow of control” in

```c
int n1 = input_from_client("Enter a number");
int n2 = input_from_client("Ok, got " + n1 + ". Now enter another");
return n1 + n2;
```

- The client answers to the page by
  1. composing an HTTP request containing values of form fields and the continuation identifier $K_1$ found in the page, and
  2. sending the request to the server.
Expected “flow of control” in

```cpp
int n1 = input_from_client("Enter a number");
int n2 = input_from_client("Ok, got " + n1 + 
    " . Now enter another");
return n1 + n2;
```

- The server handles the HTTP request by
  1. unpacking the continuation identifier \( K_1 \),
  2. looking up the associated continuation,
  3. unpacking values of form field, and
  4. invoking the continuation with the appropriate value (say 15)
Expected “flow of control” in

```java
int n1 = 15;
int n2 = input_from_client("Ok, got " + n1 + "\n\t'. Now enter another");
return n1 + n2;
```

- The server
  1. grabs the **continuation** of the second line (the second call to `input_from_client` to be precise),
  2. stores it under a identifier $K_2$,
  3. constructs a page (saying “Ok, got 15. Now enter another”) that contains the identifier $K_2$,
  4. sends the page to the client, and
  5. stops running.
The WebServer Session Simulator

> (add-webserver)
(html (submit "http://www.myserver.com/resume?k=17424")
   "Enter first number"
   ("NUMBER1" "0"))

> (service '(request "http://www.myserver.com/resume?k=17424"
               ("NUMBER1" "100")))
(html (submit "http://www.myserver.com/resume?k=89285")
   "Ok, got " 100 "Enter Second number" ("NUMBER2" "0"))

> (service '(request "http://www.myserver.com/resume?k=89285"
               ("NUMBER2" "200")))
300

> (service '(request "http://www.myserver.com/resume?k=89285"
               ("NUMBER2" "2000")))
2100

>
(define registered-continuations '())
(define (register-continuation! k)
  ;; Store continuation k under a new URL in
  ;; registered-continuations, and return that URL.
  (let ([url (format "http://www.myserver.com/resume?k=~a"
                   (random 100000))])
    (set! registered-continuations
      (cons (cons url k) registered-continuations))
    url))
(define (get-registered-continuation url)
  ;; Return the continuation stored under the given URL in
  ;; registered-continuations.
  (let ([k (assoc url registered-continuations)])
    (if k
      (cdr k)
      (error "Unknown url ~a" url))))
(define (show page-maker)
  ;; Make a page using page-maker and send the page to
  ;; the client browser. Do not return until the client
  ;; responds to the page, in which case the client is
  ;; request returned.
  (call/cc (lambda (resume)
    (let* ([resumption-url
            (register-continuation! resume)]
           [page
            (page-maker resumption-url)]
           (show-and-die page))))))
The Webserver session simulator
(Managing requests)

; REQUEST STRING -> STRING
(define (request-input-field request field-name)
   ;; Return the value of the given field in the given request.
   (cadr (assoc field-name (caddr request))))

; REQUEST -> URL
(define (request-url request)
   ;; Return the url of the given request.
   (cadr request))

; REQUEST -> NOTHING
(define (service request)
   (let ([k (get-registered-continuation (request-url request))])
     (k request)))
The Webserver session simulator
(Making the initial page)

; (UNIT -> PAGE) -> PAGE
(define (initial-request initial-page-maker)
  (call/cc (lambda (top)
    (set! top-level-continuation top)
    (initial-page-maker))))
The Webserver session simulator
(Handy functions)

; STRING * (URL -> PAGE) -> NUMBER
(define (read-number field-name page-maker)
  (let ([request (show page-maker)])
    (string->number (request-input-field request field-name)))))

; STRING -> NUMBER
(define (input-number message)
  (read-number "num" (lambda (url)
    '(html (submit ,url)
      ,message
      ("num" "0")))))

Morten Rhiger
Continuations and web servers 24/30
(define (add-webserver)
  (initial-request
   (lambda ()
     (let* ([n1 (read-number "NUMBER1"
                 (lambda (url)
                   (html (submit ,url)
                     "Enter first number"
                     ("NUMBER1" "0")))]
          [n2 (read-number "NUMBER2"
                 (lambda (url)
                   (html (submit ,url)
                     "Ok, got " ,n1
                     "Enter Second number"
                     ("NUMBER2" "0")))]
     (+ n1 n2))))))
The Webserver session simulator
(Same example, even more direct)

; () -> NUMBER
(define (add-webserver-2)
  (initial-request
   (lambda ()
     (let* ([n1 (input-number "Enter first number")]
             [n2 (input-number "Enter second number")]
     (+ n1 n2))))))
(define (login)
  (initial-request
   (lambda ()
    (let ([name (input-string "Enter login name")])
     (define (loop iterations message)
      (if (< iterations 3)
       (let ([password (input-string message)])
        (if (and (equal? name "morten")
                 (equal? password "secret"))
         "Ok, access granted"
         (loop (+ iterations 1)
              "Incorrect password. Try again")))
      "Access denied"))
    (loop 0 "Enter password"))))
Solution

- Some stored continuations may only be invoked once.
- Optimization: Once invoked, such a continuation may be garbage collected.
Representing continuations

- In CPS, continuations are represented as heap-allocated closures (representing higher-order functions)
  
  ```
  fun aux [] k = k 1
  | aux (0 :: xs) k = 0
  | aux (x :: xs) k = aux xs (fn v => k (x * v))
  ```

  ```
  fun prod xs = aux xs (fn v => v)
  ```

- First-class continuations represented as (a heap-allocated copy of) the execution stack, plus the values of all registers
We can *defunctionalize* closures to get a first-order representation of continuations:

```plaintext
datatype cont
  = INIT
  | AUX of int * cont

fun apply INIT v = v
  | apply (AUX (x, k)) v = apply k (x * v)

fun aux [] k = apply k 1
  | aux (0 :: xs) k = 0
  | aux (x :: xs) k = aux xs (AUX (x, k))

fun prod xs = aux xs INIT
```

- Morten Rhiger
- Continuations and web servers