Clicking on Delimited Continuations

`http://okmij.org/ftp/packages/caml-shift.tar.gz`
`http://okmij.org/ftp/ML/caml-web.tar.gz`

Continuation Fest 2008
Tokyo, Japan April 13, 2008

FLOLAC 2008
Taipei, Taiwan July 11, 2008

We give a light introduction to delimited continuations and unmask them in operating system context switching and input/output. Web – the interaction between a browser and a web server – let loose delimited continuations and made them clickable. That is why web programming without first-class delimited continuations is so unnatural. Conversely, the ability to capture and store delimited continuations makes coding web applications (CGI scripts) as straightforward as writing interactive console applications using read and printf, or writing a dialogue in a play. We no longer have to guess the question from an answer. We do not even need to repeat a question, letting the user repeat an answer instead (using the 'Back button').

We demonstrate the natural web programming style by writing and running live two multi-form web applications, one of which is a simple blog. We use a library of persistent delimited continuations for bytecode OCaml programs. The library also supports nested transactions. In a live demo we show that a user may repeatedly go back-and-forth between editing and previewing their blog post, perhaps in several windows. The finished post can be submitted only once.

?

Outline

- **Delimited continuations**
  - Delimited evaluation contexts, processes, breakpoints
  - Control operators shift and reset
  - A taste of formalization

Continuations and Web Services

- A simple TTY application
- CGI and the inversion of control
- Interaction and continuations
- Plain CGI scripts and persistent continuations

Web Transactions

- “Please click the Submit button only once”
- A simple blog as a TTY application
- A simple blog as a CGI application with nested transactions

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + abs( 2 * 3 ) )
```

Full context	undelimited continuation function $\text{int} \to \infty$
Partial context	delimited continuation function $\text{int} \to \text{int}$, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

This print expression is the *whole* program, which we want to run. To this end, we first *focus* (technical term) on the (sub)expression $2 * 3$ so to compute it first. If we *cut* this expression from the program, what is left is a program with the hole. The hole is the place where $2 * 3$ used to be and which we later fill with the result of evaluating $2 * 3$. The expression with the hole is called *context*. The undelimited continuation is the meaning of the context. It is a function from what me may put in the hole (integers in our case) to . . . well, the result of the whole program. This is what computed when the whole program is fully finished – and so this value is not of much interest to the program itself as the program will never get to use this value.

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + abs( 2 * 3 ) )
```

Full context	undelimited continuation function $\text{int} \to \infty$
Partial context	delimited continuation function $\text{int} \to \text{int}$, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

When the result is computed, the program is already dead. For example, we usually don't care of the value computed by our e-mail program. We are much more interested in what the e-mail program does before it finishes or dies (i.e., has it sent the message or not).

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + abs( 2 * 3 ) )
```

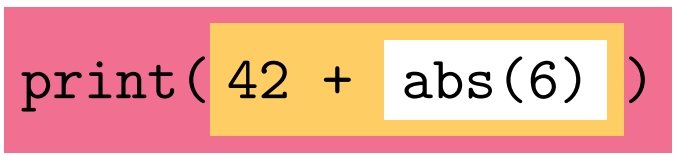
Full context	undelimited continuation function $\text{int} \to \infty$
Partial context	delimited continuation function $\text{int} \to \text{int}$, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

Beside the full context, we may also want to consider its prefix. That is, we may (mentally, for now) distinguish a subterm of a program, $42 + abs(2 * 3)$. We may imagine a boundary within print(). Taking out $2 * 3$ leaves a hole in our subterm just as it did in the whole program. This subterm with a hole is called a *partial* (evaluation) context, whose meaning is a *partial* continuation. (The subterm with a hole can be plugged into a bigger hole). The partial continuation is also a function, also from integers in our case (the type of the values that can be placed in the hole, e.g., the result of evaluating $2 * 3$). Now, however, we do care of the produced result (also called the *answer*), since we *can* do something meaningful with it: plug into a hole. So, the delimited continuation in our case is a function from int to int, namely, the function that takes an integer and adds to its absolute value 42.

Continuations are the meanings of evaluation contexts

A context is an expression with a hole



Contexts and continuations are present whether we want them or not

Let us observe what happens with the partial context as we are evaluating the term. We see the context shrinks as subterms are reduced and are replaced with values.

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + if  6>0  then 6 else neg(6) )
```

Contexts and continuations are present whether we want them or not

We also see the partial context expand when functions are invoked and their bodies are inlined.

Continuations are the meanings of evaluation contexts

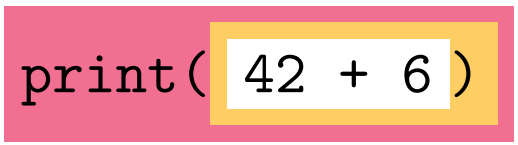
A context is an expression with a hole

```
print( 42 +  if true then 6 else neg(6)  )
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole



Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

Contexts and continuations are present whether we want them or not

Finally, our distinguished subterm is reduced to a single value and is no longer useful to distinguish it. Nothing can ever happen to 48.

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print(48)
```

Contexts and continuations are present whether we want them or not

So, the boundary and the yellow thing it envelops disappear. I apologize for the triviality of all of this. We will see more interesting examples next. We will also make the notions of disappearing boundaries and plugging of the hole precise.

Whether we are concerned with the continuations or not, they are *always* present.

(Delimited) continuations are the meanings of (delimited) evaluation contexts.

Control effects: Process scheduling in OS

Operating system, User process, System call

```
schedule( main () {... read(file) ...} ) ...
```

Let us consider a different example: the OS has invoked a user process, and the process is about to make a system call, that is, request the OS, the supervisor, to read from a given file. The slide shows the state of the whole program at this point.

Now it makes great sense to distinguish the subterm that represents the user program (in yellow) from the rest. This is the kernel-user boundary. This example also makes it clear why we usually don't care of the result of the whole program: when the OS returns a result it is because it crashed, at which point we quickly reboot.

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file) ...} ) ...
```

```
schedule( ReadRequest( PCB ,file) ) ...
```

We do one step of evaluation, and see the different picture from the one in the earlier example. All spread-out yellow stuff has disappeared in one step, replaced with this term `ReadRequest(PCB,file)`. Such a behavior is characteristic of *control effects*. But the yellow stuff is not gone, it is somehow 'saved' in the value we call here PCB, or, the process control block in the OS parlance. How the context is saved is not important for us now. We only need to know that the saved context can be restored.

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file) ...} ) ...
```

```
schedule( ReadRequest( PCB ,file) ) ...
```

...

```
schedule( resume( PCB ,"read string") ) ...
```

When the OS gets around to reading from a file, it does the following operation. The next reduction is:

Control effects: Process scheduling in OS

Capture, Invoke

```
schedule( main () {... read(file) ...} ) ...

schedule( ReadRequest( PCB ,file) ) ...

...

schedule( resume( PCB ,"read string") ) ...

schedule( main () {... "read string" ...} ) ...
```

We get the picture very similar to the original one, only with `"read string"` in place of `read(file)`.

We have seen thus the two control operations on the contexts: *capturing* a context (saving it in a value like PCB) and *restoring* it. The latter operation takes the captured context (`PCB`), a value (”read string”), plugs the value into the saved context and puts the result in the context of the restoring operation. It is easy to see that this context invocation looks exactly like the function application (cf the invocation of `abs(6)` in the first example).

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file) ...} ) ...

schedule( ReadRequest( PCB ,file) ) ...

...

schedule( resume( PCB ,"read string") ) ...

schedule( main () {... "read string" ...} ) ...
```

User-level control operations $\Rightarrow$ user-level scheduling, thread library

The operations of capturing and resuming are obviously special, here in the sense that only OS can do them. One may imagine context capturing and restoring available to user programs, too. We can then implement user-level threads and write scheduling libraries.

Captured continuation can be invoked once, none, or several times.

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1 ) )
```

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1 ) )
```

BP_1

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1 ) )
```

BP_1

```
debug_run(resume (BP1,3))
```

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1 ) )
```

BP_1

```
debug_run(resume (BP1,3))
```

```
debug_run( 42 + abs(2 * 3 ) )
```

We note that we don't have to resume from the breakpoint at all. But we did execute `resume (BP`$_1$`,3)`, which restored the context, replaced the breakpoint expression with 3, and continued running the program. Suppose we don't like the computed result, 48 in our case. We still possess the captured continuation saved as BP_1. We can resume it again, with a different value. We do normal debugging work.

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
```

Debugger is an external tool, separate from a program. Can debugging be available in the program itself? Control operators let us implement debugging: the breakpoint and `debug_run`. Let us redo the debugging session above, this time using the real OCaml code.

We use the library Delimcc of delimited control operators for bytecode OCaml. It is a *pure library*, it does not modify the OCaml system in any way. The library provides many control delimiters, called 'prompts'. The function `push_prompt` (a pseudo-special form, hence the argument is a thunk) sets the delimiter, marking the boundary of the delimited context. We use only one prompt in all our examples. In such a case it is common to regard the prompt implicit and call `push_prompt` 'reset'.

Here's our earlier expression, color-coded. The type `breakpt` describes the result of the whole yellow expression. The argument of `BP` denotes the yellow expression with the white hole; its type is `int->breakpt`. It is this delimited context that is captured by `shift`.

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))

val v1 : breakpt = BP <fun>
```

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
  val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
```

Programmable debugger

```
open Delimcc      let p0 = new_prompt ()
type breakpt = Done of int | BP of (int -> breakpt)

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
  val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
let v2 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * 3 ))
```

We note that we don't have to resume from the breakpoint at all. But we did: applying `k` to 3 restored the context, replaced the breakpoint expression with 3, and continued running the program. The restored context is enclosed in `push_prompt`.

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
let v2 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * 3 ))
val v2 : breakpt = Done 48
```

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
  val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
  val v2 : breakpt = Done 48
let v2' = let BP k = v1 in k (-5)
```

Suppose we don't like the computed result, 48 in our case. We still possess the captured continuation saved as v_1. We can resume it again, with a different value. So, we can do backtracking, answer what-if questions, and implement non-determinism.

Programmable debugger

```
open Delimcc
type breakpt = Done of int | BP of (int -> breakpt)

let p0 = new_prompt ()

let v1 = push_prompt p0 (fun () ->
    Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
  val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
  val v2 : breakpt = Done 48
let v2' = let BP k = v1 in k (-5)
let v2' = push_prompt p0 (fun () ->
    Done (42 + abs(2 * -5 ))
```

Programmable debugger

```
open Delimcc      let p0 = new_prompt ()
type breakpt = Done of int | BP of (int -> breakpt)

let v1 = push_prompt p0 (fun () ->
      Done (42 + abs(2 * shift p0 (fun k -> BP k) ))
  val v1 : breakpt = BP <fun>
let v2 = let BP k = v1 in k 3
  val v2 : breakpt = Done 48
let v2' = let BP k = v1 in k (-5)
let v2' = push_prompt p0 (fun () ->
    Done (42 + abs(2 * -5 ))
  val v2' : breakpt = Done 52
```

Debugging an iteration

```
module CSet =
    Set.Make(struct type t=char let compare=compare end)

let set1 = List.fold_right CSet.add
    ['F';'L';'O';'L';'A';'C';'O';'8']
    CSet.empty
  val set1 : CSet.t = <abstr>

CSet.iter (fun e -> print_char e) set1
  O8ACFLO
```

Set is the standard OCaml module for ordered sets, ordered collections of elements with no duplicates. Set has the usual methods: empty, add, union, intersection, etc. Among the methods is `iter`, which applies a given function to each element, in order. In our case, we build the set of characters, populate it with characters in this list. We pass `print_char` to iter, which prints each element of the set, in order. You can see the result: in order, no duplicates.

Debugging an iteration, cont

```
type cursor = EOF | Cons of char * (unit -> cursor)
let pc = new_prompt ()

let sv1 = push_prompt pc (fun () ->
   CSet.iter(fun e-> shift pc (fun k -> Cons (e,k)) ) set1;
   EOF )
   val sv1 : cursor = Cons ('0', <fun>)
```

The module `Set` is part of the standard library; we can't modify it, we can't add any new methods. We can still 'debug' it. We set the break-point in the function we pass to iter. Whenever iter invokes our function, we break on the breakpoint, and report the element given by iter.

Debugging an iteration, cont

```
type cursor = EOF | Cons of char * (unit -> cursor)
let pc = new_prompt ()

let sv1 = push_prompt pc (fun () ->
   CSet.iter(fun e-> shift pc (fun k -> Cons (e,k)) ) set1;
   EOF )
   val sv1 : cursor = Cons ('0', <fun>)

let next = function Cons (_,k) -> k ()
let sv2 = next sv1;;
   val sv2 : cursor = Cons ('8', <fun>)
let sv3 = next sv2;;
   val sv3 : cursor = Cons ('A', <fun>)
```

Debugging an iteration, cont

```
type cursor = EOF | Cons of char * (unit -> cursor)
let pc = new_prompt ()

let sv1 = push_prompt pc (fun () ->
   CSet.iter(fun e-> shift pc (fun k -> Cons (e,k)) ) set1;
   EOF )
   val sv1 : cursor = Cons ('0', <fun>)

let rec take n c = match (n,c) with
   | (0,_) | (_,EOF) -> []
   | (n,Cons (e,k)) -> e:: take (pred n) (k ())

take 5 sv2
   - : char list = ['8'; 'A'; 'C'; 'F'; 'L']
```

The interface `Set` does offer a method to get the minimal element; but no method to get the 2nd minimal, 3d minimal, etc. The function `take` lets us take the n minimal elements starting from any position – in this case, after the first minimal element.

In the first debugging example, it might appear that `call/cc` would have sufficed. The function `take` makes it clear that we really need a *delimited* continuation. When `take` invokes `(k ())` in `take (pred n) (k ())` to continue the iteration one more step, `(k ())` is expected to return a value. If the breakpoint triggered in executing (k ()) captured the whole continuation, `take` would have been captured as well. We wish to 'debug' only `iter` rather than the whole program, and we wish `take` to run our 'debugging session'.

Other applications, beside enumerator inversion: non-destructive updates (see the 'Zipper File System'), transactions.

CBN $\lambda\!\!\!\!\!\!\amalg$-calculus

Primitive Constants $D ::= \text{john} \mid \text{mary} \mid \text{see} \mid \text{tall} \mid \text{mother}$

Constants $C ::= D \mid C \wedge C \mid c \mid \forall_c \mid \partial_c$

Terms $E, F ::= V \mid x \mid FE \mid E \curlywedge F \mid Q \,\$\, E \mid \amalg k : S.\, E$

Values $V ::= C \mid u \mid \lambda x{:}T.\, E \mid W$

Strict Values $W ::= \lambda^{!} u{:}U.\, E$

Coterms $Q ::= \# \mid E, Q \mid Q ;_{!} W \mid E ,_c Q \mid Q ;_c V$

Term equalities

$Q \,\$\, FE = E, Q \,\$\, F \qquad Q \,\$\, WE = Q ;_{!} W \,\$\, E$

$Q \,\$\, F \curlywedge E = E ,_c Q \,\$\, F \qquad Q \,\$\, V \curlywedge E = Q ;_c V \,\$\, E$

$\# \,\$\, V = V$

Transitions

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, (\lambda x.\, E) F \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, E\{x \mapsto F\}$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, (\lambda^{!} x.\, E) V \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, E\{x \mapsto V\}$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, C_1 \curlywedge C_2 \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, C_1 \wedge C_2$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, Q \,\$\, \amalg k.\, E \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, \# \,\$\, E\{k \mapsto Q\}$

Earlier I was waiving hands along with bits of code. That will continue through the rest of the talk. Some say it is not a science unless it is written in Greek. I'm therefore obliged to demonstrate that delimited continuations are science. Here: lots of Greek and bizarre typography like subscripted commas. I won't describe any of that. This is a slide for my talk in a month. The slide describes a bit more general calculus: CBN with shift/reset, which embeds CBV through strict functions. The best application of the calculus seems to be linguistic, that's why our constants here are not integers but elements of a semantic domain.

Th slides shows dynamic, operational semantics. There is also a sound type system and a type-checking/Church-style reconstruction algorithm. I won't show it; it won't fit on one slide.

CBN $\lambda\!\!\!\!\!\!\amalg$-calculus

Primitive Constants $D ::= \text{john} \mid \text{mary} \mid \text{see} \mid \text{tall} \mid \text{mother}$

Constants $C ::= D \mid C \wedge C \mid c \mid \forall_c \mid \partial_c$

Terms $E, F ::= V \mid x \mid FE \mid E \curlywedge F \mid Q \,\$\, E \mid \amalg k : S.\, E$

Values $V ::= C \mid u \mid \lambda x{:}T.\, E \mid W$

Strict Values $W ::= \lambda^{!} u{:}U.\, E$

Coterms $Q ::= \# \mid E, Q \mid Q ;_{!} W \mid E ,_c Q \mid Q ;_c V$

Term equalities

$Q \,\$\, FE = E, Q \,\$\, F \qquad Q \,\$\, WE = Q ;_{!} W \,\$\, E$

$Q \,\$\, F \curlywedge E = E ,_c Q \,\$\, F \qquad Q \,\$\, V \curlywedge E = Q ;_c V \,\$\, E$

$\# \,\$\, V = V$

Transitions

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, (\lambda x.\, E)F \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, E\{x \mapsto F\}$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, (\lambda^{!} x.\, E)V \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, E\{x \mapsto V\}$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, C_1 \curlywedge C_2 \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, C_1 \wedge C_2$

$Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, Q \,\$\, \amalg k.\, E \rightsquigarrow Q_1 \,\$\, \cdots \,\$\, Q_n \,\$\, \# \,\$\, E\{k \mapsto Q\}$

Symmetry of shift and lambda: both are binding forms; one binds a variable, the other binds a co-variable. When lambda-abstraction is applied, it takes a term on the right and substitutes into the body. When shift is applied, it takes a co-term on the left and substitutes into the body.

Outline

You can see from the outline that we will be talking and *showing* that *persistent delimited* continuations are the natural fit for CGI programming. The message of this talk is that we can write CGI applications in exactly the same way we write console applications.

This talk is simple, may be too simple: it contains no theorems and no Greek symbols. It seems many in the audience are the developers, who probably won't mind seeing bits of code.

Running example, a console version

Demonstrate `test_Queinnec_tty`, the interactive console version

The main example for the first part of the talk is the example by Christian Queinnec, from his famous ICFP00 paper. This is the paper that first pointed out that web has made continuations clickable.

The application is the currency conversion, in three screens. We demonstrate the interactive console version:

```
./test_Queinnec_tty 2> /tmp/log
```

The original application used French Francs as the currency, which is no longer in circulation. I took the liberty to substitute Yen.

Running example

```
let main () =
 let henv = inquire "currency_read_rate.html" [] in
 let curr_name = answer "curr-name" henv vstring in
 let curr_rate = answer "rate" henv vfloat in
 let henv = inquire "currency_read_yen.html"
                    [("curr-name",curr_name)] in
 let amount = answer "curr-amount" henv vfloat in
 let yen_amount = amount /. curr_rate in
   inquire_finish "currency_result.html"
   [("curr-name",curr_name);("rate",string_of_float curr_rate);
    ("curr-amount",string_of_float amount);
    ("yen-amount", string_of_float yen_amount)];
   exit 0                                  (* unreachable *)
```

Templates

```
<html><head>
<title>Currency converter with respect to &yen;. Form 2</title>
</head><body>
<H1 ALIGN=CENTER>Example of (delimited) continuations on the Web

<div>${response}</div>

<form action="${this-script}" method="GET">
<input type=hidden name="klabel" value="${klabel}" size=10 maxsi
Converting ${curr-name} into &yen;.
<table>
<tr><td>Enter the amount:
<td align=right>
<input type=text name="curr-amount" value="${curr-amount}" size=
</table>
<INPUT name=submit TYPE=Submit>
</form>
</body></html>
```

Here is one of the templates.
One can write the template in any HTML or text editor, using any language, and add as many CSS or Flash animations as one wishes. In this talk, I'll keep the pages very simple.

Running example

```
let main () =
 let henv = inquire "currency_read_rate.html" [] in
 let curr_name = answer "curr-name" henv vstring in
 let curr_rate = answer "rate" henv vfloat in
 let henv = inquire "currency_read_yen.html"
                    [("curr-name",curr_name)] in
 let amount = answer "curr-amount" henv vfloat in
 let yen_amount = amount /. curr_rate in
   inquire_finish "currency_result.html"
   [("curr-name",curr_name);("rate",string_of_float curr_rate);
    ("curr-amount",string_of_float amount);
    ("yen-amount", string_of_float yen_amount)];
   exit 0                                  (* unreachable *)
```

Running example

```
let main () =
 let henv = inquire "currency_read_rate.html" [] in
 let curr_name = answer "curr-name" henv vstring in
 let curr_rate = answer "rate" henv vfloat in
 let henv = inquire "currency_read_yen.html"
                    [("curr-name",curr_name)] in
 let amount = answer "curr-amount" henv vfloat in
 let yen_amount = amount /. curr_rate in
   inquire_finish "currency_result.html"
   [("curr-name",curr_name);("rate",string_of_float curr_rate);
    ("curr-amount",string_of_float amount);
    ("yen-amount", string_of_float yen_amount)];
   exit 0                                  (* unreachable *)
```

We note the use of the lexical scope across questions. We ask a question, receive and validate the answer, bind the result to a local variable `curr_rate`, and use it later to compute the final answer.

Explain inquire in terms gettext.

Running example as a typical CGI script

```
let main () =
 let henv = get_form_env () in
 match hlocate "klabel" henv with
 | None -> send "currency_read_rate.html" []
 | Some "got-rate" ->
     (match (hlocate "curr-name" henv, hlocate "rate" henv) wit
     | (Some curr_name, (Some rate as rv)) ->
         let _ = validate rv vfloat in
         send "currency_read_yen.html"
           [("curr-name",curr_name); ("rate",rate)]
     | _ -> failwith "need error handling")
 | Some "got-amount" ->
     (match (hlocate "curr-name" henv, hlocate "rate" henv,
             hlocate "curr-amount" henv) with
     | (Some curr_name, (Some _ as rv), (Some _ as amv)) ->
         let curr_rate = validate rv vfloat in
         let amount = validate amv vfloat in
         let yen_amount = amount /. curr_rate in
         send "currency_result.html" [("curr-name",curr_name);.
     | _ -> failwith "need error handling")
 | _ -> failwith "need error handling";
```

We can re-write the console application into a CGI application. This slide shows the usual way of writing CGI scripts, with explicit continuations. We see clearly the inversion of control, the lack of lexical scope, the need for repeated validation. The inconvenience becomes even more pronounced if instead of the dispatch on the value of klabel, we split this script into three (as is common).

We note the different pattern of colors: before we were asking a question and analyzing the answers. Now, we get the answer and try to figure what the question was. All the vertical bars indicate lots of case analysis. We also see the repeated validation of `rate`.

MS-DOS vs. Mac programming: in MS DOS, we create a window, position the cursor, write a text. On a Mac, the window is created in one function, and the drawing is done in another, in response to a Draw event. Direct vs. round-about style. That's why event-based programming, although widely considered superior for high-performance distributed systems, is relatively infrequent because it is hard, especially in C and other languages with no convenient closures.

Queinnec calls this program-centric vs. page-centric. In the former, we use lexical scope rather than global variables or request objects to manage intermediate data ('rate')

Running example as a typical CGI script

```
let main () =
 let henv = get_form_env () in
 match hlocate "klabel" henv with
 | None -> send "currency_read_rate.html" []
 | Some "got-rate" ->
     (match (hlocate "curr-name" henv, hlocate "rate" henv) wit
     | (Some curr_name, (Some rate as rv)) ->
         let _ = validate rv vfloat in
         send "currency_read_yen.html"
           [("curr-name",curr_name); ("rate",rate)]
     | _ -> failwith "need error handling")
 | Some "got-amount" ->
     (match (hlocate "curr-name" henv, hlocate "rate" henv,
             hlocate "curr-amount" henv) with
     | (Some curr_name, (Some _ as rv), (Some _ as amv)) ->
         let curr_rate = validate rv vfloat in
         let amount = validate amv vfloat in
         let yen_amount = amount /. curr_rate in
         send "currency_result.html" [("curr-name",curr_name);.
     | _ -> failwith "need error handling")
 | _ -> failwith "need error handling";
```

Running example

```
let main () =
 let henv = inquire "currency_read_rate.html" [] in
 let curr_name = answer "curr-name" henv vstring in
 let curr_rate = answer "rate" henv vfloat in
 let henv = inquire "currency_read_yen.html"
                    [("curr-name",curr_name)] in
 let amount = answer "curr-amount" henv vfloat in
 let yen_amount = amount /. curr_rate in
   inquire_finish "currency_result.html"
   [("curr-name",curr_name);("rate",string_of_float curr_rate);
    ("curr-amount",string_of_float amount);
    ("yen-amount", string_of_float yen_amount)];
   exit 0                                  (* unreachable *)
```

Let us go back to the original, nice and natural console code, and look at the implementation of the basic primitives, `inquire` and `answer`.

Interaction in TTY mode

```
let do_inquire template henv =
  send_form stdout henv template;
  flush_all ();
  let henv = url_unquote_collate (read_line ()) in
  HEnv.add hvar_template_name template henv

 let inquire template outputs =
   let henv =
     List.fold_left (fun m (k,v) -> HEnv.add k v m) HEnv.empty o
   do_inquire template henv

 let inquire_finish template outputs =
   let _ = inquire template outputs in exit 0

 let answer hvar henv validfn =
   validate (fun henv ->
     do_inquire (HEnv.find hvar_template_name henv) henv)
     hvar henv validfn
```

Here is the implementation for the console version of the example. We see that `answer` does not do much: it extracts a particular answer from the set of answers `henv`, converts and validates it. Should the validation fail, `answer` repeats the question.

The real work is done by `do_inquire`, which sends the filled-in template and waits for the sequence of answers. Explain read as the capturing of a delimited continuation (user process state) by the OS. I'm obsessed in pointing out that every programmer already knows and understands the delimited continuations; they might not know that word though. Everyone knows that when a process executes a system call like 'read', it gets *suspended*. When the disk delivers the data, the process is *resumed*. That suspension of a process is its continuation. It is delimited: it is not the check-point of the whole OS, it is the check-point of a process only, from the invocation of `main()` up to the point `main()` returns. Normally these suspensions are resumed only once, but can be zero times (`exit`) or twice (`fork`).

Interaction with Continuations

```
let do_inquire template env =
  let send k =
      let klabel = gensym () in
      Res (Some (klabel,k),template,env) in
  let henv = shift p0 send in
  HEnv.add hvar_template_name template henv

let inquire template outputs =
  do_inquire template (Right outputs)

let inquire_finish template outputs =
    abort p0 (Res (None, template, Right outputs))

let answer hvar henv validfn = ... the same ...
```

Can we do something similar ourselves? Yes, if we have delimited control operators. Explain shift as a system call, `klabel` as a PID, and show a simple scheduler in the run module.

Interaction with Continuations: Main loop

```
let run () =
 let rec loop jobqueue =
   let (k,template,henv) = try
      let henv = url_unquote_collate (read_line ()) in
      let Res (k,template,env) = match
        try Some (List.assoc (HEnv.find "klabel" henv) jobqueue)
        with Not_found -> None  with
        | None -> push_prompt p0 (fun () -> main (); failwith "n
        | Some k -> k henv in
      let henv = match k with
      | Some (klabel,_) -> HEnv.add "klabel" klabel henv
      | None -> henv in
      (k,template,henv)
    with e -> base_error_handler (Printexc.to_string e) in
  print_endline "Content-type: text/html\n";
  send_form stdout henv template; flush_all ();
  match k with  | Some job -> loop (job::jobqueue)  | None -> lo
in loop []
```

Interaction with Continuations: Main loop

```
let run () =
 let rec loop jobqueue =
   let (k,template,henv) = try
      let henv = url_unquote_collate (read_line ()) in
      let Res (k,template,env) = match
        try Some (List.assoc (HEnv.find "klabel" henv) jobqueue)
        with Not_found -> None  with
        | None -> push_prompt p0 (fun () -> main (); failwith "n
        | Some k -> k henv in
      let henv = match k with
      | Some (klabel,_) -> HEnv.add "klabel" klabel henv
      | None -> henv in
      (k,template,henv)
    with e -> base_error_handler (Printexc.to_string e) in
  print_endline "Content-type: text/html\n";
  send_form stdout henv template; flush_all ();
  match k with  | Some job -> loop (job::jobqueue)  | None -> lo
in loop []
```

Interaction with Continuations: Main loop

```
let run () =
 let rec loop jobqueue =
   let (k,template,henv) = try
      let henv = url_unquote_collate (read_line ()) in
      let Res (k,template,env) = match
        try Some (List.assoc (HEnv.find "klabel" henv) jobqueue)
        with Not_found -> None  with
        | None -> push_prompt p0 (fun () -> main (); failwith "n
        | Some k -> k henv in
      let henv = match k with
      | Some (klabel,_) -> HEnv.add "klabel" klabel henv
      | None -> henv in
      (k,template,henv)
    with e -> base_error_handler (Printexc.to_string e) in
  print_endline "Content-type: text/html\n";
  send_form stdout henv template; flush_all ();
  match k with  | Some job -> loop (job::jobqueue)  | None -> lo
in loop []
```

Interaction with Continuations: Main loop

```
let run () =
 let rec loop jobqueue =
   let (k,template,henv) = try
      let henv = url_unquote_collate (read_line ()) in
      let Res (k,template,env) = match
        try Some (List.assoc (HEnv.find "klabel" henv) jobqueue)
        with Not_found -> None  with
        | None -> push_prompt p0 (fun () -> main (); failwith "n
        | Some k -> k henv in
      let henv = match k with
      | Some (klabel,_) -> HEnv.add "klabel" klabel henv
      | None -> henv in
      (k,template,henv)
    with e -> base_error_handler (Printexc.to_string e) in
  print_endline "Content-type: text/html\n";
  send_form stdout henv template; flush_all ();
  match k with  | Some job -> loop (job::jobqueue)  | None -> lo
in loop []
```

Interaction with Continuations: Main loop

```
let run () =
 let rec loop jobqueue =
   let (k,template,henv) = try
      let henv = url_unquote_collate (read_line ()) in
      let Res (k,template,env) = match
        try Some (List.assoc (HEnv.find "klabel" henv) jobqueue)
        with Not_found -> None  with
        | None -> push_prompt p0 (fun () -> main (); failwith "n
        | Some k -> k henv in
      let henv = match k with
      | Some (klabel,_) -> HEnv.add "klabel" klabel henv
      | None -> henv in
      (k,template,henv)
    with e -> base_error_handler (Printexc.to_string e) in
  print_endline "Content-type: text/html\n";
  send_form stdout henv template; flush_all ();
  match k with  | Some job -> loop (job::jobqueue)  | None -> lo
in loop []
```

A primitive OS scheduler that almost fits into one slide. At the beginning of the loop we do `read_line` – wait for something to happen, for data to arrive. In other words, wait for an interrupt.

But this isn't CGI. These in-memory delimited continuations are suitable for a persistent server (FastCGI). But a CGI script *dies* after the interaction. We need to find a way to survive the death. That is not as impossible as it sounds. We need to make the captured continuation persistent.

Interaction with Persistent Continuations

```
type cgi_result' =
   Res of string * (henv,(hvar * string) list) either
type cgi_result = unit -> cgi_result'
type k_t = henv -> cgi_result

let do_inquire template env =
 let sendk (k : k_t) () =
  let fname  = Filename.temp_file "kcgi" "" in
  let klabel = Filename.basename fname in
  let () = save_state_file fname (Obj.repr k) in
  Res (template,add "klabel" klabel env) in
 let henv = shift p0 sendk in
 HEnv.add hvar_template_name template henv
```

Here is the implementation of the basic primitives using persistent delimited continuations.

Interaction with Persistent Continuations

```
type cgi_result' =
   Res of string * (henv,(hvar * string) list) either
type cgi_result = unit -> cgi_result'
type k_t = henv -> cgi_result

let do_inquire template env =
 let sendk (k : k_t) () =
  let fname  = Filename.temp_file "kcgi" "" in
  let klabel = Filename.basename fname in
  let () = save_state_file fname (Obj.repr k) in
  Res (template,add "klabel" klabel env) in
 let henv = shift p0 sendk in
 HEnv.add hvar_template_name template henv
```

Interaction with Persistent Continuations

```
type cgi_result' =
   Res of string * (henv,(hvar * string) list) either
type cgi_result = unit -> cgi_result'
type k_t = henv -> cgi_result

let do_inquire template env =
 let sendk (k : k_t) () =
  let fname  = Filename.temp_file "kcgi" "" in
  let klabel = Filename.basename fname in
  let () = save_state_file fname (Obj.repr k) in
  Res (template,add "klabel" klabel env) in
 let henv = shift p0 sendk in
 HEnv.add hvar_template_name template henv
```

Main Loop with Persistent Continuations

```
let run () =
  let (template,henv) =
    try
      let henv = get_form_env () in
      let Res (template,env) =
        match
          try Some (locate_cont (HEnv.find "klabel" henv))
            with Not_found -> None | Sys_error _ -> None with
          | None -> push_prompt our_p0 (fun () -> main (); failw
          | Some k -> k henv () in
        (template,henv)
      with e -> base_error_handler (Printexc.to_string e) in
    print_endline "Content-type: text/html\n";
    send_form stdout henv template; flush_all ();
    exit 0
```

The implementation is almost the same as above. The file system is now the jobqueue. That is how we gain persistence.

Rather than storing the continuation on disk, we could have serialized it in a string and included in the web form: `fname` would have been the value of the file rather than the name of the file with the stored continuation. In that case, the Web itself becomes our 'job queue'.

Main Loop with Persistent Continuations

```
let run () =
  let (template,henv) =
    try
      let henv = get_form_env () in
      let Res (template,env) =
        match
          try Some (locate_cont (HEnv.find "klabel" henv))
            with Not_found -> None | Sys_error _ -> None with
          | None -> push_prompt our_p0 (fun () -> main (); failw
          | Some k -> k henv () in
        (template,henv)
      with e -> base_error_handler (Printexc.to_string e) in
    print_endline "Content-type: text/html\n";
    send_form stdout henv template; flush_all ();
    exit 0
```

Demo

The back button and the multiple windows

`ls -lt /tmp/kcgi*`, note the timestamps and the sizes of saved continuations

Demonstrate the CGI script. Show the benefit of explicit (rather than OS-captured) continuations: the back button. When going back to the 'amount' slide and forth, note that the rate information was stored in the continuation. We really has kept our state across death.

Show parallel conversions with two different rates (in two different windows). Although intermediate pages (asking for the amount) look the same, they correspond to different continuations, embody different rates, and so produce different results.

Do `ls -lt /tmp/kcgi*` Note the timestamps of the files (it was the live test indeed) and the sizes of the stored continuations. Bzip2 can compress them twice. Describe the benefit of being delimited: only part of the state is saved.

Stored continuations constitute the trail. We can re-display computation from each of these points; There are already timestamps; we can also record for each k what url brought us there (in fact, this information is already present). If this currency conversion being an exercise, we can see how a student wandered through it. The student can see, too, and go back and forth in their exploitation of the material. Queinnec talks more about trails and exploration of an interactive course in his ICFP00 paper.

Running example

```
let main () =
 let henv = inquire "currency_read_rate.html" [] in
 let curr_name = answer "curr-name" henv vstring in
 let curr_rate = answer "rate" henv vfloat in
 let henv = inquire "currency_read_yen.html"
                    [("curr-name",curr_name)] in
 let amount = answer "curr-amount" henv vfloat in
 let yen_amount = amount /. curr_rate in
   inquire_finish "currency_result.html"
   [("curr-name",curr_name);("rate",string_of_float curr_rate);
    ("curr-amount",string_of_float amount);
    ("yen-amount", string_of_float yen_amount)];
   exit 0                                  (* unreachable *)
```

The demo we have just seen used this code – exactly the same code that was used in the console application at the beginning of the talk. We *can* indeed write CGI code as if it were a console application. We merely use a different implementation of the i/o primitives.

Advantages of delimited continuations

Persistent continuations are twice delimited – in control and data

- makes them small
- makes them *possible*
- makes them correct

The thread-local scope ('thread+offspring') arises naturally and requires no implementation

Ease of use

- Unmodified OCaml
- Unmodified web server (e.g., Apache)
- vs. custom Java-based CPS Scheme interpreter and web server

Queinnec wrote a special Scheme interpreter and a web server in Java. The interpreter is written in CPS with defunctionalized continuations. Here, we use the standard, *unmodified* OCaml. We also use the standard, unmodified web server (e.g., Apache and w3m). We write everything in direct style.

Describe 'twice delimited' – capture not only part of the stack but also part of the global closures on the heap referred to from the stack. Like moving noodles from one bowl to another (one closure pulls the other). We need to cut the noodles, or closures. When serializing the code pointer, we don't serialize the code itself. Likewise, we don't serialize all of the global data (In his extended paper, Queinnec complains about the sizes of stored continuations, which store all global data).

Emphasize the novelty: just serializing the captured continuation won't work (abstract data type `_chan`); and if it worked, it wouldn't have been correct (global data structures of the delimcc library itself would have been captured). Thus serialization is quite a bit more complex than it appears. The audience is the first to see serialized delimited continuations in OCaml.

Another benefit of delimited continuations: make natural what Queinnec calls a new thread+offspring scope – thread-local dynamic binding – or delimited dynamic binding, described in our ICFP06

Outline

Show `payment.html` in a web browser.

Discuss the problem: you pressed ”commit” and the reply page never came because of some network error. Does it mean that the form never got submitted (and so the credit card has not been charged and I have to try again), or that the form did get submitted but the reply is lost (and so the credit card was charged and I should NOT try again). There is no way of telling short of waiting and calling the credit card company.

Design of a simple blog

```
let main () =
  let henv = inquire "blog_login.html" [] in
  let username = answer "username" henv vstring in
  let ()       = answer "password" henv ...
  let rec loop_browse () =
    let content = read_blog () in
    let henv = inquire "blog_view.html" (("blog-data",content)::
    if answer "logout" henv vbool then inquire_finish "blog_logo
    else
      if not (answer "new" henv vbool) then loop_browse () else
      let henv = inquire "blog_new.html" env in
      let rec loop_edit henv =
        if answer "cancel" henv vbool then loop_browse () else
        let title = answer "title" henv vstring in
        let body  = answer "body" henv vstring in
        let new_post = markup username title body in
        if answer "submit" henv vbool then
          let () = write_blog new_post in loop_browse ()
        else let henv = inquire "blog_new.html" ([("title",title
          loop_edit henv
    in loop_edit henv
```

Design of a simple blog

```
let main () =
  let henv = inquire "blog_login.html" [] in
  let username = answer "username" henv vstring in
  let ()       = answer "password" henv ...
  let rec loop_browse () =
    let content = read_blog () in
    let henv = inquire "blog_view.html" (("blog-data",content)::
    if answer "logout" henv vbool then inquire_finish "blog_logo
    else
      if not (answer "new" henv vbool) then loop_browse () else
      let henv = inquire "blog_new.html" env in
      let rec loop_edit henv =
        if answer "cancel" henv vbool then loop_browse () else
        let title = answer "title" henv vstring in
        let body  = answer "body" henv vstring in
        let new_post = markup username title body in
        if answer "submit" henv vbool then
          let () = write_blog new_post in loop_browse ()
        else let henv = inquire "blog_new.html" ([("title",title
          loop_edit henv
      in loop_edit henv
```

Note the nested loop

Demo of the blog

1. Login
2. Enter a new article (subject ‘Takao-san’, text ‘great hike’), submit
3. Enter another article (subject ‘Summit’, text ‘many people’)
4. Preview, go back, edit (place ’!’ in the body), preview, optionally go back, edit again, finally submit
5. Go back to one of the previous pages of editing and previewing the second article. An attempt to press any of the buttons brings the main screen. The second article, once submitted, cannot be resubmitted
6. Duplicate the window (tab)
7. In one window, enter a new article (subject ‘Way back’, text ‘slow’), preview, don’t submit
8. In the other tab, enter a new article (subject ‘Nature course’, text ‘narrow, dark, wonderful’), preview, submit, logout
9. Go back to the first tab still previewing another article. An attempt to submit brings back the login screen: the closed outer transaction invalidates all inner ones

Simple blog as a transactional CGI script

```
let rec main () =
  let henv = inquire "blog_login.html" [] in
  let username = answer "username" henv vstring in
  let ()       = answer "password" henv ...
  let env = [("username",username)] in
  let edit env = ... in
  try
  in_transaction env (fun env ->        (* user session tx *)
    let rec loop_browse () =
      let content = read_blog () in
      let henv = inquire "blog_view.html" (("blog-data",content)
      if answer "logout" henv vbool then ()
      else if not (answer "new" henv vbool) then loop_browse ()
      match (try in_transaction env edit with TX_Gone -> None) w
      | None -> loop_browse ()
      | Some new_post -> write_blog new_post; loop_browse ()
    in loop_browse ());
  inquire_finish "blog_logout.html" env
  with TX_Gone -> main ()
```

Simple blog as a transactional CGI script

```
let rec main () =
  let henv = inquire "blog_login.html" [] in
  let username = answer "username" henv vstring in
  let ()       = answer "password" henv ...
  let env = [("username",username)] in
  let edit env = ... in
  try
  in_transaction env (fun env ->        (* user session tx *)
    let rec loop_browse () =
      let content = read_blog () in
      let henv = inquire "blog_view.html" (("blog-data",content)
      if answer "logout" henv vbool then ()
      else if not (answer "new" henv vbool) then loop_browse ()
      match (try in_transaction env edit with TX_Gone -> None) w
      | None -> loop_browse ()
      | Some new_post -> write_blog new_post; loop_browse ()
    in loop_browse ());
  inquire_finish "blog_logout.html" env
  with TX_Gone -> main ()
```

The new post editing function

```
let edit env =
  let henv = inquire "blog_new.html" env in
  if answer "cancel" henv vbool then None else (* rollback *)
  let title = answer "title" henv vstring in
  let body  = answer "body" henv vstring in
  let new_post = markup username title body in
  if answer "submit" henv vbool then Some new_post (* commit *)
  else
    let () = assert (answer "preview" henv vbool) in
    let henv = inquire "blog_preview.html" (("new-post",new_post
    if answer "submit" henv vbool then Some new_post (* commit *
    else None
```

CGI: no loop in editing. Show the outer transaction (logged in user) and the nested editing transaction. Show what it means to commit and rollback.

`in_transaction` could use delimited env (dynamic binding), so all the labels etc are dynamically bound.

Conclusions

First implementation of persistent twice-delimited continuations in OCaml

Persistent delimited continuations are the natural fit for CGI programming

CGI script $\equiv$ console application

with differently implemented IO primitives

- natural dialogue
- lexical scoping, exception handling
- mutable data, if necessary

Delimited continuations are concrete and clickable

Main lesson: Delimited continuations are quite tangible after all. Every time you click on a submit button, you're clicking on a delimited continuation.