# Delimited Control in OCaml, Abstractly and Concretely

## System Description

http://okmij.org/ftp/Computation/Continuations.html

FLOPS 2010
Sendai, Japan    April 21, 2010

## Outline

- **Applications**

Introduction to delimited continuations

Implementation

The subject of the talk the library of multi-prompt delimited control in OCaml. The library is called delimcc.

# Delimited dynamic binding

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)
Amr Sabry (Indiana University)

ICFP 2006

The first application of the library of multi-prompt delimited continuations in OCaml was implementing delimited dynamic binding, presented at ICFP 2006. This is the title slide from that talk, given by Chung-chieh Shan.

# Demo of persistent delimited continuations in OCaml for nested web transactions

`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

A practical application of the delimcc library was to automagically turn console applications into CGI applications. You merely link the application with a different IO library. This process was demonstrated at the Continuation Fest in Toukyou two years and a week ago. Again, this was the title slide from that talk.

# Shifting the stage
# Staging with delimited control

Yukiyoshi Kameyama and Oleg Kiselyov and Chung-chieh Shan

PEPM, 20 January 2009

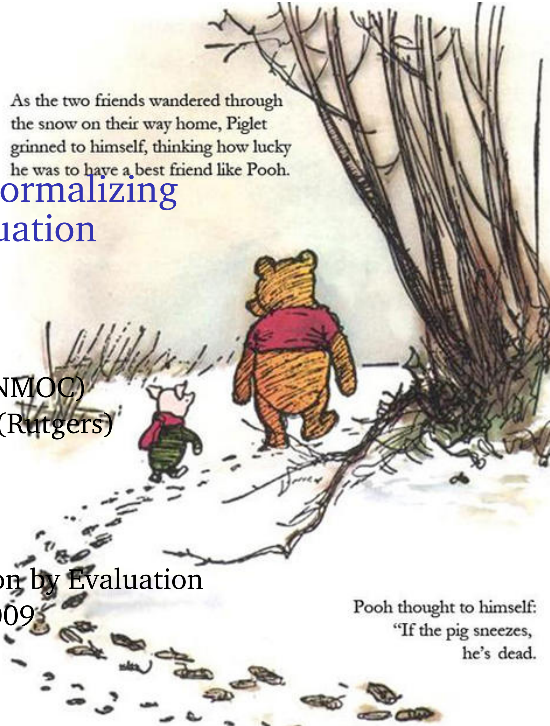
The library turned out useful for writing efficient and comprehensible direct-style code generators.

# Monolingual probabilistic programming using generalized coroutines

Oleg Kiselyov and Chung-chieh Shan

Uncertainty in Artificial Intelligence (UAI)
McGill University, 19 June 2009

Another, quite extensive application was a very shallow embedding of a probabilistic domain-specific language. This is the first slide from that talk, first presented less than a year ago by Chung-chieh Shan, at a quite well-known AI conference.


As the two friends wandered through the snow on their way home, Piglet grinned to himself, thinking how lucky he was to have a best friend like Pooh.
Pooh thought to himself: "If the pig sneezes, he's dead.

# Lifted inference: normalizing loops by evaluation

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers)

Workshop on Normalization by Evaluation
15 August 2009

We have found another application, normalization of MapReduce-loop bodies by evaluation. Chung-chieh Shan presented that talk at the LICS-affiliated workshop on normalization by evaluation. The ostensible motivation was computing probabilities of getting swine flu.

# Functional un|unparsing

Kenichi Asai (Ochanomizu)
Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers)

MitchFest, Northeastern University, Boston
23 August 2009

The type-safe printf and scanf are already available in OCaml, via ad-hoc typing rules in the OCaml type checker. We *derived* versions that don't require such ad hoc extensions to the type system. Hindley-Milner system suffices.

## Dynamic Logic in ACG: discourse anaphora and scoping islands

Logical Methods for Discourse
Nancy, December 15, 2009

Delimited continuations are useful in linguistics; one may argue that multi-prompt delimited control of the sort implemented in the delimcc library is even more useful. This talk made this argument, using OCaml code to demonstrate sample analyses of English sentences.

I hoped to convince you that the library of delimited control in OCaml is quite useful, at least for writing papers. None of the above papers said anything about the library itself or its implementation. If you are curious how it is implemented, you had to read the comments in the code. There are a lot of them. Reading the comments is still a good idea: the part about persistence of the delimited continuation is not described in the FLOPS paper at all, due to space limit.

## Outline

Applications

▶ **Introduction to delimited continuations**

Implementation

This FLOPS paper is about implementing the library. Before we get to the implementation, we should remind ourselves what delimited continuations are. I emphasize the word ‘remind’ since it is my contention that everyone already knows delimited continuations.

## Puzzle

”U2” has a concert that starts in 17 minutes and they must all cross a bridge to get there. They stand on the same side of the bridge. It is night. There is one flashlight. A maximum of two people can cross at one time, and they must have the flashlight with them. The flashlight must be walked back and forth. A pair walk together at the rate of the slower man’s pace:

| | |
|---|---|
| Bono | 1 minute to cross |
| Edge | 2 minutes to cross |
| Adam | 5 minutes to cross |
| Larry | 10 minutes to cross |

For example: if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.

Allegedly, this is a question for potential Microsoft employees. An answer is expected within 5 minutes.

There are two answers, neither of which are trick answers. Allegedly, this is one of the questions for potential Microsoft employees. Some people really get caught up trying to solve this problem. Reportedly, one guy solved it by writing a C program, although that took him 37 minutes to develop (compiled and ran on the 1st try though). Another guy solved it in three minutes. A group of 50, at Motorola, couldn’t figure it out at all.

## Simple library for non-determinism

```
module type SimpleNonDet = sig
   val choose : 'a list -> 'a
   val run : (unit -> unit) -> unit
end

let fail () = choose []
```

A clear and elegant way of solving the puzzles like ours is to use non-determinism. We assume non-deterministic functions with the simple interface above. We assume that a non-deterministic computation will print out its result when it finishes. Therefore, its return type, and the return type of run are both unit. The convenient function fail fails the computation.

## Solving the puzzle

```
type u2 = Bono | Edge | Adam | Larry
type side = u2 list

let rec loop trace forward time_left = function
  | ([], _) when forward  ->
      print_trace (List.rev trace)
  | (_, []) when not forward  -> ...
  | (side_from, side_to) ->
      let party   = select_party side_from in
      let elapsed = elapsed_time party in
      let _ = if elapsed > time_left then fail () in
      let side_from' = without party side_from in
      let side_to'   = side_to @ party in
      loop ((party,forward)::trace) (not forward)
           (time_left - elapsed) (side_to',side_from')
```

This code represents the specification of the problem, in the most straightforward way. I’m sure everyone in the audience can write this code in their sleep. Perhaps only one function would give a pause.

## Selecting a party

```
let select_party side =
  let p1 = choose side in
  let p2 = choose side in
  match compare p1 p2 with
  | 0 -> [p1]
  | n when n < 0 -> [p1;p2]
  | _ -> fail ()
```

### Running the code to solve the puzzle

```
run (fun () ->
  loop [] true 17 ([Bono;Edge;Adam;Larry],[]))
```

But even the selection function is most straightforward, if we could non-deterministically select an element from the list. And our simple library provides exactly that function. The library also gives us a run function, to execute the computation.

## Implementing non-determinism

```
let rec choose = function
    | []     -> exit 666
    | [x]    -> x
    | (h::t) ->
         let pid = fork () in
         if pid = 0 then h
         else wait (); choose t

let run m = match fork () with
    | 0 -> m (); printf "Solution found"; exit 0
    | _ -> try while true do waitpid [] 0 done
           with ...
```

One way to implement non-determinism is just to run all the choices, perhaps in parallel, and hope one of them eventually succeeds. At the point of making a choice, we split the computation into several parts. Each split-off computations proceed with one of the choices. Everyone here knows how to split the computations: use fork.

It indeed works. It is interesting to watch, using top, how processes are launched and how they die, how their number increases and drops.

## Implementing non-determinism

```
let rec choose = function
    | []      -> exit 666
    | [x]     -> x
    | (h::t) ->
          let pid = fork () in
          if pid = 0 then h
          else wait (); choose t

let run m = match fork () with
    | 0 -> m (); printf "Solution found"; exit 0
    | _ -> try while true do waitpid [] 0 done
           with ...
```

I'd like to point out the `fork` in `run`: we split the computation into a process that does all the work, and the supervisor. As in real life, the supervisor immediately goes to sleep. It wakes up when all the workers are finished, to report the achieved result or an exception.

Of course the implementation is slow: Unix processes are quite heavy-weight. We need lighter processes: green threads, so to speak. We need a *green fork*.

## Green non-determinism

```
open Delimcc

let p = new_prompt ()


let choose xs = shift p (fun k -> List.iter k xs)


let run m = push_prompt p m
```

And here is the green implementation. The function shift is like fork. The latter returns to the parent a pid. In contrast, shift returns to the parent the representation `k` of the child process, as a function. The child process is suspended. (Please take the body of shift, `List.iter k xs`, as the parent computation and the computation where shift appears, the 'outside', as the child computation.) When the parent applies k to a value, the child process is resumed with that value. These difference between shift and fork are superficial, right? The function `push_prompt` too splits the computation, creating the worker that executes m, and the supervisor that waits, handles failures and gets the result. In a sense, `push_prompt` is like the try block. The prompt is akin to a communication channel, which the child process uses to tell the supervisor of its final result or exception.

## Outline

Applications

Introduction to delimited continuations

▶ **Implementation**

## Highlights of `delimcc`

- It is a byte-code *library*
  - no changes to OCaml compiler/runtime
  - perfect source- and binary- compatibility
- Direct
  - no code transformations
  - only the needed continuation prefix is captured
  - fully integrates with native exceptions
- General
  - don't mess with the stack
  - extensible to other languages
- An informally justified implementation of the formally justified abstract machine

Our delimcc is a library, making no changes to the compiler or the run-time system of OCaml. Therefore, it is perfectly compatible with the existing source code and even already compiled byte-code. It is direct in that it captures only the needed part of the control stack. The implementation is general. We don't mess with the stack (introducing new frames or changing stack frames to mark them). We use OCaml's own operations for stack manipulation. Since we consciously avoid being tied to the structure of the stack, the implementation can be extended to the languages other than OCaml.

The implementation is not fully formally derived, and no Coq was used. The correctness argument cannot be formal: after all, there is no formal specification of OCaml, with or without delimited control. The library implements an abstract machine that is formally justified by the definitional machine.

## Generality and justification

- Start with the definitional machine
- Formally transform to a form suitable for implementation
- Derive scAPI, minimalistic API for low-level stack manipulation
- Determine how OCaml byte-code machine implements scAPI already
- Implement delimcc in OCaml code, using scAPI via FFI

We outline the process by which the library was half-way–formally derived. The details are in the paper.

## scAPI

```
type ek
type ekoff
type ekfragment

val get_ek : unit -> ek
val add_ek : ek -> ekoff -> ek
val sub_ek : ek -> ek -> ekoff
val pop_stack_fragment : ek -> ek -> ekfragment
val push_stack_fragment : ekfragment -> unit
```

- *No* operations to scan the stack for a particular frame
- The format of the stack is unknown
- Porting delimcc to a different language $\equiv$ porting scAPI
- delimcc in Scheme; memory-efficient shift/reset in Scheme

Here is the scAPI. We have abstract types describing a mark on a stack (ek), a fragment of the stack between two marks, and the offset between two marks. We need operations to extract a fragment of the stack between two marks and to put the fragment on the top of the stack. There are *no* operations to scan the stack looking for a particular frame.

Porting delimcc to a different language is essentially figuring out how that other language could implement scAPI; the rest is automatic. As an illustration, the delimcc distribution shows how delimcc can be ported to Scheme. Specializing that implementation to one prompt gives a new implementation of the ordinary shift/reset in Scheme. It is different from that by Danvy and Filinski: our shift always captures only the needed prefix of the whole continuation, even though it relies on call/cc.

## Continuations and exceptions

```
let test2_ex lst =
  let f x acc =  if x = 0 then raise Zero else x * acc in
  let rec loop acc = function
    | []   -> acc
    | h::t -> try f h (loop acc t) with Other -> -1
  in
  try loop 1 lst with Zero -> 0
```

What is this stack mark? How can we get hold of it if we don't even know the structure of the stack? Let me use the following benchmark code from the delimcc distribution to describe marks. The goal is to return the product of the elements in a list of integers. Once we encountered a zero, we can return the result immediately. To this end, we throw an exception. The (contrived) code illustrates installing handlers for other exceptions. They are transparently skipped over by our exception `Zero`.

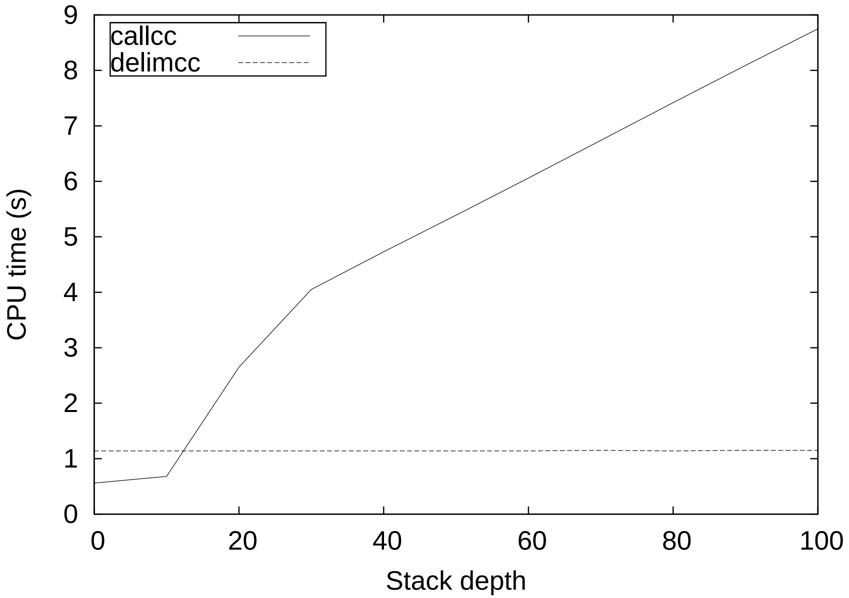
## Continuations and exceptions

```
let test2_ex lst =
  let f x acc =  if x = 0 then raise Zero else x * acc in
  let rec loop acc = function
    | []   -> acc
    | h::t -> try f h (loop acc t) with Other -> -1
  in
  try loop 1 lst with Zero -> 0
```

```
let test2_abort lst =
  let p  = new_prompt () in let p’ = new_prompt () in
  let f x acc =  if x = 0 then abort p 0 else x * acc in
  let rec loop acc = function
    | []   -> acc
    | h::t -> push_prompt p’ (fun () -> f h (loop acc t))
  in
  push_prompt p (fun () -> List.fold_right f lst 1)
```

We can write the code using delimcc. The two implementations have exactly the same run-time performance, no matter whether zero appears near the beginning or the end of the list. The function `push_prompt` is really the try block. The paper spends a great deal explaining that the mark is the identity of an exception frame; hence the name `ek` used in scAPI. The marked frames are created by the OCaml run-time itself, when the try block is entered. In general, a language system that has exception handling is implementing a part of scAPI already. The paper even has a formal argument about it.

## Undelimited vs delimited continuations



This graph illustrates the other part of scAPI. There exists a library of *undelimited* continuations for OCaml, providing callcc. Capturing an undelimited continuation copies the whole stack. The callcc library distribution comes with a co-routine benchmark. We can invoke the benchmark either at the top-level (stack depth 0), or from a non-tail recursive function that called itself 10, 20, etc. 100 times. The graph plots the running time vs. the stack depth. We also re-implemented the benchmark using delimited continuations. Again, we plot the running time vs. the depth of the stack at the time the benchmark was invoked. In either case, the benchmark creates two co-routines, which invoke each other. We need to only capture the continuation of the current co-routine to the start of the benchmark. The delimcc implementation does exactly that; it doesn't copy the whole stack. To actually copy a part of the stack we use OCaml's own mechanism to copy the stack to re-size it.

## Conclusions

### Abstract and concrete implementations of delimited control in OCaml

- Concrete: `delimcc`
- Abstract: minimalistic scAPI; formally relating exception handling to delimited control

### delimcc as an existence proof

- efficient implementation
- non-invasive implementation
- in a typed language
- in a language designed without regard to continuation passing
- no compiler plug-ins
- no run-time extensions beyond the basic FFI

We have presented abstract and concrete implementations of multi-prompt delimited control. The concrete implementation is the delimcc OCaml library, which has been fruitfully used for over four years. The abstract implementation has related delimited control to exception handling and distilled scAPI, a minimalistic API, sufficient for the implementation of delimited control. A language system accommodating exception handling and stack-overflow recovery is likely to support scAPI. The OCaml byte-code does support scAPI, and thus permits, *as it is*, the implementation of delimited control. We described the implementation of delimcc as an example of using scAPI in a typed language. This library shows that delimited control can be implemented efficiently (without copying the whole stack) and non-invasively in a typed language that was not designed with delimited control in mind and that offers no compiler plug-ins or run-time extensions beyond a basic foreign-function interface.

## The idea to remember

`shift` is a