Sequentialising a concurrent program using continuation-passing style

Juliusz Chroboczek
Université de Paris-Diderot (Paris 7)
jch@pps.jussieu.fr

28 January 2012

Outline

Everything you ever wanted to know but were afraid to ask about:

- event-driven programming;
- continuation-passing style (CPS) transform.

(Side-effect: crash course in Scheme.)

Outline

Everything you ever wanted to know but were afraid to ask about:

- event-driven programming;
- continuation-passing style (CPS) transform.

(Side-effect: crash course in Scheme.)

Everything I always wanted to tell you about:

- sequentialising threaded programs into event-driven style ;
- Continuation Passing C (CPC).

Implementations of concurrency

There are at least two techniques for writing concurrent programs:

- threads;
- event-driven programming.

What is the relationship between the two?

Implementations of concurrency

There are at least two techniques for writing concurrent programs:

- threads;
- event-driven programming.

What is the relationship between the two?

Conclusion

Threaded programs can be translated into event-driven programs by performing a partial CPS transform.

This can be done

- by hand (this tutorial), or
- automatically (CPC, joint work with Gabriel Kerneis).

Augmented Scheme

We work in Scheme augmented with three functions:

- (`ding`): plays a sound.
- (`current-time`): returns the current (monotonic) time, in seconds;
- (`sleep-until` *time*): does nothing until the given (monotonic) time.

We suppose that `sleep` can be implemented as:

```
(define (sleep delta)
  (sleep-until (+ (current-time) delta)))
```

Augmented Scheme (2)

Possible implementation in *Racket*:

```
(define (ding)
  (play-sound "ding.wav" #t))

(define (current-time)
  (/ (current-inexact-milliseconds) 1000.0))

(define (sleep-until t)
  (let loop ()
    (if (< (current-time) t)
        (loop)
        #f)))
```

This is not quite correct: real time is not monotonic time.

A trivial problem

A trivial problem: play a sound every 1/2 s (2 Hz).
First try:

```
(define (periodic)
  (ding)
  (sleep 0.5)
  (periodic))
```

Incorrect: assumes that `(ding)` takes no time to execute, will accumulate skew.

A trivial problem (2)

A trivial problem: play a sound every 1/2 s (2 Hz).
Correct version:

```
(define (periodic)
  (let loop ((start (current-time)))
    (ding)
    (sleep-until (+ start 0.5))
    (loop (+ start 0.5))))
```

Note: this is syntactic sugar for

```
(define (periodic)
  (letrec ((loop (lambda (start)
                   ...)))
    (loop (current-time))))
```

A not-quite-trivial problem

A not-quite-trivial problem: play sounds at 2 Hz and 3 Hz simultaneously.
Obvious idea: use threads.

```
(define (periodic hz)
  (let loop ((start (current-time)))
    (ding)
    (sleep-until (+ start (/ hz)))
    (loop (+ start (/ hz)))))

(define (two-hands)
  (thread (lambda () (periodic 2)))
  (thread (lambda () (periodic 3))))
```

Avoiding threads

There are reasons to want to avoid threads:

- you're using a programming language with no support for threads (Javascript or C-64 BASIC);
- you're on an embedded system and cannot afford multiple stacks;
- your program requires tens of thousands of concurrent tasks (web server), and you cannot afford that many stacks;
- you're giving a tutorial about avoiding threads.

Avoiding threads

There are reasons to want to avoid threads:

- you're using a programming language with no support for threads (Javascript or C-64 BASIC);
- you're on an embedded system and cannot afford multiple stacks;
- your program requires tens of thousands of concurrent tasks (web server), and you cannot afford that many stacks;
- you're giving a tutorial about avoiding threads.

We want a solution that avoids threads while being language-agnostic:

- no first-class continuations.

A thread-less solution

A not-quite-trivial problem: play sounds at 2 Hz and 3 Hz simultaneously, without using threads.

A thread-less solution

A not-quite-trivial problem: play sounds at 2 Hz and 3 Hz simultaneously, without using threads.

Compute the times at which a sound must be played:

left	0		0.5		1		...
right	0	0.3333		0.6667	1	1.3333	...
merged	0	0.3333	0.5	0.6667	1	1.3333	...

Note that the merged stream is 1-periodic.

A thread-less solution

A not-quite-trivial problem: play sounds at 2 Hz and 3 Hz simultaneously, without using threads.

Compute the times at which a sound must be played:

left	0		0.5		1		...
right	0	0.3333		0.6667	1	1.3333	...
merged	0	0.3333	0.5	0.6667	1	1.3333	...

Note that the merged stream is 1-periodic.

(Where did I cheat?)

A thread-less solution (3)

```
(define (ding-loop times)
  ;; takes a sorted list of times
  (cond
    ((null? times) #f)
    (#t
     (let ((when (car times)))
       (sleep-until when)
       (ding))
     (ding-loop (cdr times)))))

(define (merge l1 l2)
  ;; merge two sorted lists
  ...
)
```

A thread-less solution (4)

We can now say:

```
(define (periodic-list freq)
  (let loop ((first 0))
    (if (<= first 1)
        (cons first (loop (+ first (/ freq))))
        '())))

(define (two-hands)
  (let* ((left (periodic-list 2))
         (right (periodic-list 3))
         (merged (merge left right)))
    (let loop ()
      (let ((start (current-time)))
      (ding-loop
        (map (lambda (t) (+ start t)) merged))
      (loop))))
```

A thread-less solution (5)

```
(define (two-hands)
  (...
    (let loop ()
      (let ((start (current-time)))
      (ding-loop
        (map (lambda (t) (+ start t)) merged))
      (loop))))
```

There is slight delay between the end of `ding-loop` and the next call to `current-time` is not zero. This delay accumulates.

Exercice: implement `two-hands` in a way that doesn't accumulate delay.

A non-trivial problem

The previous solution relies on the stream of events being periodic. That doesn't generalise.

A non-trivial problem: play sounds at π Hz and e Hz simultaneously, without using threads.

The resulting stream is no longer periodic.

A non-trivial problem

The previous solution relies on the stream of events being periodic. That doesn't generalise.

A non-trivial problem: play sounds at π Hz and e Hz simultaneously, without using threads.

The resulting stream is no longer periodic.

Idea: use infinite lists (streams).

0	0.3333	0.5	0.6667	1	1.3333	1.5	1.6667	...

A non-trivial problem (2)

With lazy lists (streams), we could write a fully general solution.

```
(define (periodic-stream freq)
  ;; returns an infinite list
  (let loop ((first 0))
    (cons-lazy first (loop (+ first (/ freq))))))

(define (two-hands f1 f2)
  (ding-loop
   (map-lazy (lambda (t) (+ start t))
             (merge (periodic-stream f1)
                    (periodic-stream f2)))))
```

A non-trivial problem (3)

Fairly natural in Haskell:

```
ding :: IO ()
get_time :: IO Float
sleep_until :: Float -> IO ()
merge :: Ord a => [a] -> [a] -> [a]

two_hands :: Float -> Float -> IO ()
two_hands f1 f2 = do
  start <- get_time
  mapM_ (\t -> sleep_until (start + t) >> ding)
        (merge [0, 1/f1..] [0, 1/f2..])
```

Streams with closures

In general, the actions associated with each timer are not all identical.

Associate a closure (event handler) with each timer (event specification).

0	0	0.3333	0.5	0.6667	1	1	...
ding	ding	ding	ding	ding	ding	ding	...

Streams with closures (2)

Recall the main loop with simple streams :

```
(define (ding-loop times)
  ;; takes a sorted list of times
  (cond
    ((null? times) #f)
    (#t
     (let ((when (car times)))
       (sleep-until when)
       (ding))
     (ding-loop (cdr times)))))
```

Streams with closures (3)

With closures, we now have a generic main loop:

```
(define main-loop (handlers)
  (cond
    ((null? handlers) #f)
    (#t
     (let* ((handler (car handlers))
            (when (car event))
            (what (cdr event)))
       (sleep-until when)
       ((what)))
     (main-loop (cdr handlers)))))
```

Event-driven programming

Idea: make the list global and mutable.

Consequence: the event handlers can mutate the list of handlers.

Event-driven programming (2)

0	0			
ding insert(0.5)	ding insert(0.3333)			

	0		0.5	
	ding insert(0.33)		ding insert(1)	

		0.33	0.5	
		ding insert(0.67)	ding insert(1)	

			0.5	0.67
			ding insert(1)	ding insert(1)

Event-driven programming (3)

We need some infrastructure to maintain the list of scheduled event handlers:

```
;; A sorted list of event handlers
(define handlers '())

(define (insert-handler h hs)
  ;; insert handler h at the right spot in list hs
  ;; return the list
  ...
)

(define (insert-handler! when what)
  ;; insert a new handler at the right spot
  ;; in the global handlers list
  (let ((h (cons when what)))
    (set! handlers
          (insert-handler h handlers))))
```

Event-driven programming (4)

The event loop is in charge of executing any scheduled handlers:

```
(define (event-loop)
  (cond
   ((null? handlers) #f)
   (#t
    (let* ((event (car handlers))
           (when (car event))
           (what (cdr event)))
      (set! handlers (cdr handlers))
      (sleep-until when)
      ((what)))
    (event-loop))))
```

Note that order is important.

Event-driven programming (5)

Schedule event handlers from the event handlers themselves.

```
(define (periodic-handler start freq)
  (let ((next (+ start (/ freq))))
    (ding)
    (insert-handler!
     next (lambda () (periodic next freq)))))

(define (periodic)
  (periodic-handler (current-time) 2)
  (event-loop))
```

Event-driven programming (6)

Remember `two-hands`?

```
(define (two-hands)
  (thread (lambda () (periodic 2)))
  (thread (lambda () (periodic 3))))
```

Exercice: implement `two-hands` in event-driven style.

Event-driven programming (6)

Remember `two-hands`?

```
(define (two-hands)
  (thread (lambda () (periodic 2)))
  (thread (lambda () (periodic 3))))
```

Exercice: implement `two-hands` in event-driven style.

```
(define (two-hands f1 f2)
  (let ((start (current-time)))
    (periodic-handler start f1)
    (periodic-handler start f2))
  (main-loop))
```

Three programming techniques

We have seen three programming techniques:

- elementary sequential programming doesn't compose or doesn't generalise;
- threads require heavy-weight infrastructure;
- event-driven programming breaks the flow of control.

Idea: automatic transformation from threads to events.
This is a partial CPS transform!

Continuation Passing Style

Intuitively, the continuation of a program fragment is "what remains do be done".

For example, in

```
(begin
  (display "A")
  (display "B")
  (display "C"))
```

The continuation of

```
(display "A\n")
```

is

```
(begin
  (display "B")
  (display "C"))
```

Continuation Passing Style (2)

In Continuation Passing Style (CPS), every function is called with an explicit continuation:

```
(begin
  (display "A")
  (display "B"))
```

becomes

```
(display* "A"
  (lambda ()
    (display* "B"
      (lambda () #f))))
```

Continuation Passing Style (3)

Similarly,

```
(begin
  (display "A")
  (display "B")
  (display "C"))
```

becomes

```
(display* "A"
  (lambda ()
    (display* "B"
      (lambda ()
        (display* "C"
          (lambda () #f))))))
```

Continuation Passing Style (4)

What does the function `display*` look like?
A possible implementation:

```
(define display* (thing k)
  (display thing)
  (k))
```

Continuation Passing Style (4)

What does the function `display∗` look like?
A possible implementation:

```
(define display∗ (thing k)
  (display thing)
  (k))
```

Constraint: CPS functions can only be called with an empty dynamic chain (in “hereditary tail position”).

The following is not allowed:

```
(begin
  (display∗ "A" (lambda () #f))
  (display∗ "B" (lambda () #f))
```

Partial CPS

A CPS need not be total: it is possible to only CPS transform parts of the program.

```
(begin
  (display "A")
  (display "B")
  (display "C"))
```

can become

```
(begin
  (display "A")
  (display∗ "B"
    (lambda () (display∗ "C" (lambda () #f)))))
```

or even

```
(begin
  (display "A")
  (display "B")
  (display∗ "C" (lambda () #f)))
```

Partial CPS (2)

The constraint:

Constraint: CPS functions can only be called with an empty dynamic chain (in “hereditary tail position”).

implies the following constraint:

Constraint: a CPS function may only be called by another CPS function, not by a direct-style function.

(On the other hand, a CPS function may call a direct-style function.)

CPS version of sleep-until

We define `sleep-until*`, the CPS version of `sleep-until`:

```
(define (sleep-until* time k)
  (sleep-until time)
  (k))
```

Constraint: `sleep-until*` may only be called with an empty dynamic chain (in “hereditary tail position”).

This constraint is what makes `insert-event!` a valid implementation of `sleep-until*`.

Partial CPS (1)

Transform our first program so that it uses `sleep-until*`.

```
(define (periodic)
  (let loop ((start (current-time)))
    (ding)
    (sleep-until (+ start 0.5))
    (loop (+ start 0.5))))
```

Partial CPS (2)

```
(define (periodic)
  (let loop ((start (current-time)))
    (ding)
    (sleep-until (+ start 0.5))
    (loop (+ start 0.5))))
```

Remove syntactic sugar, rename `loop` to `periodic-handler`:

```
(define (periodic)
  (letrec ((periodic-handler
             (lambda (start)
               (ding)
               (sleep-until (+ start 0.5))
               (periodic-handler (+ start 0.5)))))
    (periodic-handler (current-time))))
```

Partial CPS (3)

```
(define (periodic)
  (letrec ((periodic-handler
            (lambda (start)
              (ding)
              (sleep-until (+ start 0.5))
              (periodic-handler (+ start 0.5)))))
    (periodic-handler (current-time))))
```

Lift the function `periodic-handler`:

```
(define (periodic-handler start)
  (ding)
  (sleep-until (+ start 0.5))
  (periodic-handler (+ start 0.5)))

(define (periodic)
  (periodic-handler (current-time)))
```

Partial CPS (4)

```
(define (periodic-handler start)
  (ding)
  (sleep-until (+ start 0.5))
  (periodic-handler (+ start 0.5)))

(define (periodic)
  (periodic-handler (current-time)))
```

CPS-convert any function that calls `sleep-until`:

```
(define (periodic-handler* start k)
  (ding)
  (sleep-until (+ start 0.5))
  (periodic-handler* (+ start 0.5) k))

(define (periodic* k)
  (periodic-handler* (current-time) k))
```

Partial CPS (5)

```
(define (periodic-handler* start k)
  (ding)
  (sleep-until (+ start 0.5))
  (periodic-handler* (+ start 0.5) k))

(define (periodic* k)
  (periodic-handler* (current-time) k))
```

We can now convert all calls to `sleep-until` into calls to `sleep-until*`:

```
(define (periodic-handler* start k)
  (ding)
  (sleep-until* (+ start 0.5))
    (lambda () (periodic-handler* (+ start 0.5) k))

(define (periodic* k)
  (periodic-handler* (current-time) k))
```

Partial CPS (5)

```
(define (periodic-handler∗ start k)
  (ding)
  (sleep-until∗
    (+ start 0.5))
    (lambda () (periodic-handler∗ (+ start 0.5) k))
```

Except for the useless parameter k, this is almost exactly our hand-written event-driven code:

```
(define (periodic-handler start freq)
  (let ((next (+ start (/ freq))))
    (ding)
    (insert-handler!
     next (lambda () (periodic next freq)))))
```

A non-trivial continuation

Exercice: convert the following code into event-driven style by performing a partial CPS.

```
(define (wait-a-sec)
  (let ((start (current-time)))
    (sleep-until (+ start 1))))

(define (ding-ding)
  (ding)
  (wait-a-sec)
  (ding))
```

A non-trivial continuation (2)

Solution:

```
(define (wait-a-sec* k)
  (let ((start (current-time)))
    (sleep-until* (+ start 1) k)))

(define (ding-ding* k)
  (ding)
  (wait-a-sec* (lambda () (ding) (k))))
```

In this case, the continuation cannot be optimised away without some more work.

Continuation Passing C

Continuation Passing C (CPC) is an automatic translator from threaded to event-driven code based on a partial CPS.

The target language is C, which complicates matters:

- no closures: use lambda-lifting (correct in this particular case, even though C is a cbv imperative language);
- variable capture (& operator): boxing of a small number of variables
- no closures: continuations are implemented using an ad hoc data structure.

Continuation Passing C

```
#include "cpc/cpc_runtime.h"

cps void ding(void);

cps void
periodic(double hz)
{
    while(1) {
        ding();
        cpc_sleep(1.0/hz);
    }
}

int
main()
{
    cpc_spawn{ periodic(2); }
    cpc_spawn{ periodic(3); }
    cpc_main_loop();
}
```

Conclusion

Event-driven programming is just performing a partial CPS and optimising it on the fly. In your head.

CPC

CPC (joint work with Gabriel Kerneis) is an automated translator that uses the technique outlined above to convert C with threads into plain sequential C.

`http://www.pps.jussieu.fr/~kerneis/software/cpc/`

Acknowledgements

Thanks to Thibaut Balabonski and Gabriel Kerneis for the video.