Sequentialising a concurrent program using continuation-passing style

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

6 September 2011

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 talk), 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 (R^nRS Scheme 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've been asked to give a "Distilled Tutorial" at DSL'2011.

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 (3)

```
(define (ding-list times)
  ;; takes a sorted list of times
  (cond
    ((null? times) #f)
    (#t
     (sleep-until (car times))
     (ding)
     (ding-list (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-list
        (map (lambda (t) (+ start t)) merged))
      (loop)))))
```

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).

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-list
   (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..])
```

Event-driven programming

The more usual (and more general?) technique is called *event-driven programming*.

Event: something that can trigger an action to be taken, e.g. a timer firing.

Event handler: a pair of an event specification and code to execute when a matching event triggers.

We maintain a (finite) list of previously scheduled event handlers.
The event-loop is in charge of executing the correct handler when an event matches.

Event-driven programming (2)

For our example, event handlers will be of the form

(cons *time action*)

We will maintain a sorted finite list of previously scheduled event handlers:

time	0	0.3333	0.5
action	`ding`	`ding`	`ding`

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
  ...
)

(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))))
```

Event-driven programming (5)

We can now say:

```
(define (brillig)
  (let ((start (current-time)))
    (insert-handler!
     (+ start 0.7)
     (lambda ()
      (display "and the slithy toves...\n"))))
    (insert-handler!
     (+ start 0.3)
     (lambda ()
      (display "'Twas brillig "))))))
  (main-loop))
```

...which is not particularly exciting.

Event-driven programming (6)

More exciting: 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)
  (main-loop))
```

Event-driven programming (7)

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 (7)

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

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

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

A CPS function is only called with an empty dynamic chain (in “hereditary tail position”).

Continuation Passing Style (2)

```
(display "’Twas brillig ")
(display "and the slithy toves...\n")
```

Continuation Passing Style (2)

```
(display "'Twas brillig ")
(display "and the slithy toves...\n")
```

becomes

```
(display*
 "'Twas brillig "
 (lambda ()
  (display* "and the slithy toves...\n"
   (lambda () #f))))
```

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.

Conclusion

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

Une page de publicité

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.