

Schachtelbare Engines

Schachtelbare Engines

In der Praxis: Threads starten neue Threads

Mit bisheriger Implementierung nicht möglich

Schwierigkeit: Mehrere Engines können gleichzeitig aktiv sein

Zeitaufteilung

Die Zeit zwischen einer Engine und ihren Vorgängern wird *fair verteilt*:

Jeder Tick wird sowohl der Engine als auch ihren Vorgängern berechnet

Aktive Engines

Es können mehrere Engines gleichzeitig laufen.

Die globalen Variablen `active?`, `do-return` und `do-expire` verwalten immer nur eine Engine

Stattdessen: Verwalte die laufenden Engines mit einem Stack

Zuletzt gestarte Engine ist oberstes Element auf dem Stack

Stack implementieren

```
(define stack '())

(define (push . l)
  (set! stack (cons l stack)))

(define (pop handler)
  (if (null? stack)
      (error "attempt to return from inactive engine")
      (let ((top (car stack)))
        (set! stack (cdr stack))
        (apply handler top))))

(define (active?)
  (not (null? stack)))
```

Engine erzeugen

- Engines sind noch immer Prozeduren
- Code zur Verwaltung des Engine-Stacks in `run`
- Momentane Engine anhalten, neue laufen lassen

```
(define (new-engine resume)
  (lambda (ticks return expire)
    ((call/cc
      (lambda (escape)
        (run resume
          (stop-timer)
          ticks
          (lambda (value ticks)
            (escape (lambda () (return value ticks))))
          (lambda (engine)
            (escape (lambda () (expire engine)))))))))))
```

Engines verwalten

- Ein Stack-Frame ist eine Liste mit `parent-ticks`, `child-ticks`, `return` und `expire`
- Laufzeit berechnen und Engine starten

```
(define (run resume parent-ticks child-ticks return expire)
  (let ((ticks (if (and (active?)
                       (< parent-ticks child-ticks))
                  parent-ticks
                  child-ticks)))
    (push (- parent-ticks ticks)
          (- child-ticks ticks)
          return
          expire)
    (resume ticks)))
```

Timer setzen

- Timer-Handler ruft `do-expire` mit aktueller Continuation als Argument auf
- Timer mit Restzeit neu starten

```
(define (go ticks)
  (if (active?)
      (if (zero? ticks)
          (timer-handler)
          (start-timer ticks timer-handler))))

(define (timer-handler)
  (go (call/cc do-expire)))
```

return

- Die momentan laufende Engine soll zurückkehren
- Die `return`-Prozedur und die Timing-Informationen sind im obersten Stack-Frame

```
(define (do-return value ticks)
  (pop (lambda (parent-ticks child-ticks return expire)
        (go (+ parent-ticks ticks))
          (return value (+ child-ticks ticks))))))

(define (engine-return value)
  (do-return value (stop-timer)))
```

expire

- Der abgelaufene Timer betrifft nicht unbedingt die zuletzt gestartete Engine
- D. h. die abgelaufene Engine ist nicht unbedingt oberstes Element des Stacks
- Wird die abgelaufene Engine neu gestartet: Nachfolger ebenfalls starten

```
(define (do-expire resume)
  (pop (lambda (parent-ticks child-ticks return expire)
        (if (> child-ticks 0)
            (do-expire (lambda (ticks)
                          (run resume ticks child-ticks
                                return expire))))
          (begin
              (go parent-ticks)
              (expire (new-engine resume)))))))
```

Echtes Thread-System

Engines können als Grundlage für ein Thread-System verwendet werden (Beispiel Scheme 48):

Unterschiede:

- Verwende Timer vom Betriebssystem
- *poll*: Überprüfen, ob Timer abgelaufen
- Poll-Aufrufe werden automatisch vom Compiler eingefügt

Zusammenfassung

- Thread-System mit schachtelbaren Threads implementiert
- Keine Erweiterung der Sprache notwendig
- Continuations sind die Grundlage
- Threads sind leicht: Berechnung mit einer Continuation festgehalten
- Komplizierte Scheduler denkbar