

CML

- Nebenläufige Spracherweiterung
- Verwendet Message-Passing
- Nachrichtenaustausch über Channels
- Synchrones Senden und Empfangen

CML in scsh

Laden mit

```
scsh -le1 cml/load.scm -o threads -o rendezvous -o rendezvous-channels
```

Homepage enthält (bald) Bauanleitung für daheim

- (**make-channel**) liefert (bidirektionalen) Channel
- (**receive ch**) liest Wert aus Channel (synchron)
- (**send ch v**) sendet Wert an Channel (synchron)

Synchrone Operationen

Betrachte `send` auf synchronen Channels:

- Beschreibt ein Ereignis auf einem synchronen Channel ("Sende Nachricht auf Channel")
- Wartet auf den Eintritt dieses Ereignisses ("Synchronisation")

Analog für `receive`:

- Beschreibt ein Ereignis auf einem synchronen Channel ("Empfange Nachricht auf Channel")
- Wartet auf den Eintritt dieses Ereignisses

⇒ `send` und `receive` erfüllen jeweils zwei Aufgaben

Aber: Ereignis ist bis jetzt implizit und damit nicht greifbar

Rendezvous

CML macht diese Ereignisse als eigene Werte verfügbar: sog. *Rendezvous*

- (`send-rv ch v`) beschreibt das Ereignis "sende `v` auf Kanal `ch`"), liefert Rendezvous
- (`receive-rv ch`) beschreibt das Ereignis "empfangen Wert von Kanal `ch`"), liefert Rendezvous

CML bietet Möglichkeiten, um

- auf Rendezvous zu warten
- Rendezvous zu kombinieren

Warten auf Rendezvous

(`sync rv`) wartet auf den Eintritt (die Synchronisation) eines Rendezvous

Gibt den Wert der Kommunikation zurück: *Rendezvous-Wert*

Synchronisation auf ein Rendezvous ist beliebig oft möglich!

Definition von send und receive

Beide Aufgaben zusammenfassen:

`(send ch v) ≡ (sync (send-rv ch v))`

`(receive ch) ≡ (sync (receive-rv ch))`

Aktion nach Synchronisation: wrap

`(wrap rv f)` assoziiert eine Funktion `f` mit einem Rendezvous `rv`.

- Funktion wird ausgeführt, wenn Rendezvous synchronisiert wird
- Funktion bekommt Rendezvous-Wert übergeben und liefert neuen Wert
- Ergebnis von `wrap` ist wieder ein Rendezvous

```
(sync (wrap (receive-rv ch)
            (lambda (v)
              (+ v 1))))
```

Selektive Kommunikation

Häufiges Problem: Thread muss auf mehrere Ereignisse gleichzeitig warten

- Prozessnetzwerke bei denen die Knoten mehrere Ein- oder Ausgänge haben
- Server hört auf mehreren Kanälen

```
(define (add ch-1 ch-2 res)
  (spawn
    (lambda ()
      (let lp ()
        (send res (+ (receive ch-1)
                     (receive ch-2))))
      (lp))))))
```

Problem: Gefahr von Deadlock, wenn anderer Thread in "falscher" Reihenfolge an `ch-1` und `ch-2` sendet

select

Lösung: `(select rv1 ...)` wartet, bis auf eines der Rendezvous `rv1 ...` synchronisiert werden kann, gibt dann dessen Rendezvous-Wert zurück

```
(define (add ch-1 ch-2 res)
  (spawn
    (lambda ()
      (select
        (wrap (receive-rv ch-1)
              (lambda (v)
                (send res (+ v (receive ch-2))))))
        (wrap (receive-rv ch-2)
              (lambda (v)
                (send res (+ v (receive ch-1))))))))))
```

choose

Beobachtung: `select` übernimmt auch wieder zwei Aufgaben:

- Ereignis beschreiben
- Auf Ereignis synchronisieren

⇒ Primitive Form `choose` beschreibt das Ereignis
"nicht-deterministische Auswahl"

Liefert Rendezvous

`(select rv1 ...)` \equiv `(sync (choose rv1 ...))`

Beispiel für Kombination von Rendezvous

`choose` kombiniert mehrere Rendezvous zu neuem Rendezvous

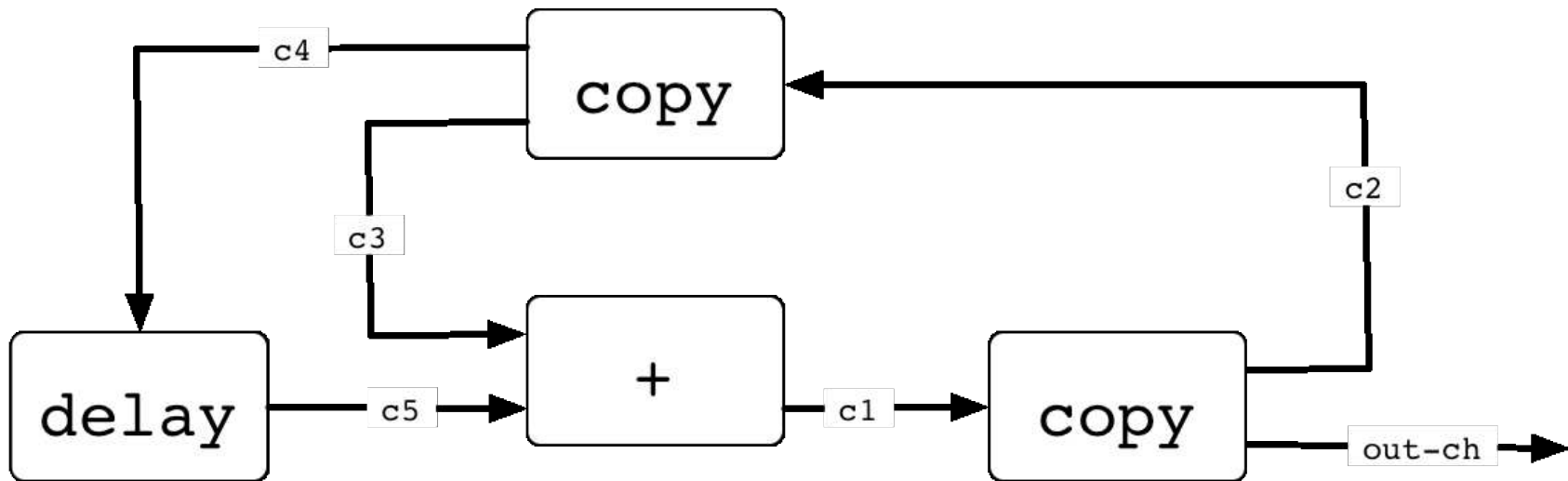
```
(define complex-rv
  (choose (wrap rv1 proc1)
          (wrap (choose (wrap rv2 proc2)
                        rv3)
              proc4)))
```

Beispiel: Fibonacci-Funktion

Definition als rekursive Gleichungen:

- $\text{fib } 1 = 1$
- $\text{fib } 2 = 1$
- $\text{fib } (i+2) = \text{fib } i + \text{fib } (i+1)$

Fibonacci als Prozessnetzwerk



copy-Knoten

Kopiere Wert in 2 Ausgabechannel

- Endlosschleife
- Senden in beide Ausgabechannel gleichzeitig probieren

```
(define (copy in-ch out-ch1 out-ch2)
  (spawn
    (lambda ()
      (let lp ()
        (let ((v (receive in-ch)))
          (select
            (wrap (send-rv out-ch1 v)
                  (lambda (ignore)
                    (send out-ch2 v)))
            (wrap (send-rv out-ch2 v)
                  (lambda (ignore)
                    (send out-ch1 v))))))
        (lp))))))
```

delay-Knoten

Gleichzeitig probieren:

- Empfange Wert
- Gib letzten Wert aus

```
(define (delay init in-ch out-ch)
  (spawn
    (lambda ()
      (let lp ((v init))
        (select
          (wrap (send-rv out-ch v)
                (lambda (ignore)
                  (lp (receive in-ch))))))
          (wrap (receive-rv in-ch)
                (lambda (next)
                  (send out-ch v)
                  (lp next))))))))))
```

Das Netzwerk

Erst Channels (= Kanten) dann Knoten erzeugen

```
(define (make-fib-network)
  (let ((out-ch (make-channel))
        (c1 (make-channel))
        (c2 (make-channel))
        (c3 (make-channel))
        (c4 (make-channel))
        (c5 (make-channel)))
    (delay 0 c4 c5)
    (copy c2 c3 c4)
    (add c3 c5 c1)
    (copy c1 c2 out-ch)
    (send c1 1)
    out-ch))
```

Beispiel: Zellen mit CML

Server empfängt Anfragen aus 2 Kanälen

(vorher: 2 Anfragetypen)

```
(define-record-type :cell
  (really-make-cell ref-channel set!-channel)
  cell?
  (ref-channel cell-ref-channel)
  (set!-channel cell-set!-channel))
```

Server für Zelle

Verwende `select`, um aus beiden Kanälen lesen zu können

```
(define (make-cell value)
  (let ((ref-channel (make-channel))
        (set!-channel (make-channel)))
    (spawn
      (lambda ()
        (let loop ((value value))
          (select
            (wrap (send-rv ref-channel value)
                  (lambda (ignore)
                    (loop value)))
            (wrap (receive-rv set!-channel)
                  loop))))))
    (really-make-cell ref-channel set!-channel)))
```

Clienten für Zellen

Senden an passenden Channel:

```
(define (cell-ref cell)
  (receive (cell-ref-channel cell)))

(define (cell-set! cell value)
  (send (cell-set!-channel cell) value))
```

guard

(`guard f`)

- liefert Rendezvous
- wendet `f` vor Synchronisation an, `f` liefert Rendezvous
- Synchronisation auf Rendezvous von `f`

Also Aktion vor Synchronisation (anstatt danach wie bei `wrap`)

Anwendung: Ressourcen bereitstellen

Swap-Channels

Channel, über den Threads Werte austauschen

Jeder Thread sendet und empfängt einen Wert

Idee: Wer Nachricht sendet, schickt auch Channel für die Antwort

```
(define-record-type :swap-channel
  (really-make-swap-channel channel)
  swap-channel?
  (channel swap-channel-channel))
```

```
(define (make-swap-channel)
  (really-make-swap-channel (make-channel)))
```

Swap-Channels

`swap` liefert Rendezvous, auf das synchronisiert werden kann

Für jede Synchronisation ist neuer Antwortchannel nötig

⇒ Verwende `guard` um Antwortchannel zu erzeugen

```
(define (swap-rv swap-channel message-out)
  (let ((channel (swap-channel-channel swap-channel)))
    (guard
      (lambda ()
        (let ((in-channel (make-channel)))
          (choose
            (wrap (receive-rv channel)
                  (lambda (pair)
                    (let ((message-in (car pair))
                          (out-channel (cdr pair)))
                      (send out-channel message-out)
                      message-in))))
            (wrap (send-rv channel (cons message-out in-channel))
                  (lambda (ignore)
                    (receive in-channel))))))))))
```

Swap-Channels in Aktion

```
(define (make-const-reply x ch)
  (let ((rv (swap-rv ch x)))
    (let lp ()
      (display (sync rv))
      (newline)
      (lp))))
(define c1 (make-swap-channel))
(spawn (lambda ()
         (make-const-reply 42 c1)))
(spawn (lambda ()
         (make-const-reply 23 c1)))
```

liefert 42, 23, 42, 23, 42, 23,...

`guard` ist wichtig, damit bei jedem `sync` ein neuer Antwortchannel generiert wird

Bedeutung von Rendezvous

Rendezvous entspricht der Beschreibung einer Kommunikation

Rendezvous können vielfältig kombiniert werden (**wrap**, **choose**, **guard**)

⇒ Bei neuen Abstraktionen stets Rendezvous verfügbar machen

Zusammenfassung

- Senden und Empfangen lassen sich noch weiter zerlegen
- Rendezvous sind die Bausteine für komplexere Abstraktionen
- Selektive Kommunikation hilft Deadlocks zu vermeiden