

Script generated by TTT

Title: Seidl: Info2 (26.01.2018)

Date: Fri Jan 26 08:33:05 CET 2018

Duration: 92:28 min

Pages: 29

Allgemeiner

$\text{app} (\text{rev } x) y = \text{rev1 } x y$ für alle Listen x, y .

Beweis: Induktion nach der Länge n von x

$n = 0$: Dann gilt: $x = []$. Wir schließen:

```
app (rev x) y = app (rev []) y
              = app (match [] with [] -> [] | ...) y
              = app [] y
              = y
              = match [] with [] -> y | ...
              = rev1 [] y
              = rev1 x y
```

353

Beispiel 2

```
let rec rev = fun x -> match x
  with [] -> []
      | h::t -> app (rev t) [h]
let rec rev1 = fun x -> fun y -> match x
  with [] -> y
      | h::t -> rev1 t (h::y)
```

Behauptung

$\text{rev } x = \text{rev1 } x []$ für alle Listen x .



352

$n > 0$: Dann gilt: $x = h::t$ wobei t Länge $n - 1$ hat.

Wir schließen (unter Weglassung einfacher Zwischenschritte):

```
app (rev x) y = app (rev (h::t)) y
              = app (app (rev t) [h]) y
              = app (rev t) (app [h] y) wegen Beispiel 1
              = app (rev t) (h::y)
              = rev1 t (h::y) nach Induktionsvoraussetzung
              = rev1 (h::t) y
              = rev1 x y
```

354

Diskussion

- Wieder haben wir implizit die Terminierung der Funktionsaufrufe von `app`, `rev` und `rev1` angenommen.
- Deren Terminierung können wir jedoch leicht mittels Induktion nach der Tiefe des ersten Arguments nachweisen.
- Die Behauptung:

folgt aus:

$$\begin{aligned} \text{rev } x &= \text{rev1 } x \ [] \\ \text{app } (\text{rev } x) \ y &= \text{rev1 } x \ y \end{aligned}$$

indem wir: $y = []$ setzen und Aussage (1) aus [Beispiel 1](#) benutzen.

355

Behauptung

$\text{sorted } x \wedge \text{sorted } y \rightarrow \text{sorted } (\text{merge } x \ y)$
für alle Listen x, y .

Beweis: Induktion über die **Summe** n der Längen von x, y .

Gelte $\text{sorted } x \wedge \text{sorted } y$.

$n = 0$: Dann gilt: $x = [] = y$

Wir schließen:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &= \text{sorted } (\text{merge } [] \ []) \\ &= \text{sorted } [] \\ &= \text{true} \end{aligned}$$

357

Beispiel 3

```
let rec sorted = fun x -> match x
  with h1::h2::t -> (match h1 <= h2
    with true -> sorted (h2::t)
      | false -> false)
  | _ -> true
```

```
and merge = fun x -> fun y -> match (x,y)
  with ([],y) -> y
  | (x,[]) -> x
  | (x1::xs,y1::ys) -> (match x1 <= y1
    with true -> x1 :: merge xs y
      | false -> y1 :: merge x ys)
```

356

Beispiel 3

```
let rec sorted = fun x -> match x
  with h1::h2::t -> (match h1 <= h2
    with true -> sorted (h2::t)
      | false -> false)
  | _ -> true
```

```
and merge = fun x -> fun y -> match (x,y)
  with ([],y) -> y
  | (x,[]) -> x
  | (x1::xs,y1::ys) -> (match x1 <= y1
    with true -> x1 :: merge xs y
      | false -> y1 :: merge x ys)
```

356

Behauptung

$\text{sorted } x \wedge \text{sorted } y \rightarrow \text{sorted } (\text{merge } x \ y)$
für alle Listen x, y .

Beweis: Induktion über die **Summe** n der Längen von x, y .

Gelte $\text{sorted } x \wedge \text{sorted } y$.

$n = 0$: Dann gilt: $x = [] = y$

Wir schließen:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &= \text{sorted } (\text{merge } [] \ []) \\ &= \text{sorted } [] \\ &= \text{true} \end{aligned}$$

357

Fall 3: $x = x1::xs \wedge y = y1::ys \wedge x1 \leq y1$.

Wir schließen:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &= \text{sorted } (\text{merge } (x1::xs) \ (y1::ys)) \\ &= \text{sorted } (x1 \ :: \ \text{merge } xs \ y) \\ &= \dots \end{aligned}$$

Fall 3.1: $xs = []$

Wir schließen:

$$\begin{aligned} \dots &= \text{sorted } (x1 \ :: \ \text{merge } [] \ y) \\ &= \text{sorted } (x1 \ :: \ y) = \text{sorted } (x1 \ :: \ y1::y) \\ &= \text{sorted } y = \text{sorted } (y1 \ :: \ ys) \\ &= \text{true} \end{aligned}$$

359

$n > 0$:

Fall 1: $x = []$.

Wir schließen:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &= \text{sorted } (\text{merge } [] \ y) \\ &= \text{sorted } y \\ &= \text{true} \end{aligned}$$

Fall 2: $y = []$ analog.

358

Fall 3.2: $xs = x2::xs' \wedge x2 \leq y1$.

Insbesondere gilt: $x1 \leq x2 \wedge \text{sorted } xs$.

Wir schließen:

$$\begin{aligned} \dots &= \text{sorted } (x1 \ :: \ \text{merge } (x2::xs') \ y) \\ &= \text{sorted } (x1 \ :: \ x2 \ :: \ \text{merge } xs' \ y) \\ &= \text{sorted } (x2 \ :: \ \text{merge } xs' \ y) \\ &= \text{sorted } (\text{merge } xs \ y) \\ &= \text{true} \text{ nach Induktionsannahme} \end{aligned}$$

360

Fall 3: $x = x1::xs \wedge y = y1::ys \wedge x1 \leq y1$.

Wir schließen:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (x1 :: merge xs y)
                  = ...
```

Fall 3.1: $xs = []$

Wir schließen:

```
... = sorted (x1 :: merge [] y)
    = sorted (x1 :: y)
    = sorted y
    = true
```

359

Fall 4: $x = x1::xs \wedge y = y1::ys \wedge x1 > y1$.

Wir schließen:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (y1 :: merge x ys)
                  = ...
```

Fall 4.1: $ys = []$

Wir schließen:

```
... = sorted (y1 :: merge x [])
    = sorted (y1 :: x)
    = sorted x
    = true
```

362

Fall 3.2: $xs = x2::xs' \wedge x2 \leq y1$.

Insbesondere gilt:

$x1 \leq x2 \wedge \text{sorted } xs$.

Wir schließen:

```
... = sorted (x1 :: merge (x2::xs') y)
    = sorted (x1 :: x2 :: merge xs' y)
    = sorted (x2 :: merge xs' y)
    = sorted (merge xs y)
    = true nach Induktionsannahme
```

360

Fall 4.2: $ys = y2::ys' \wedge x1 > y2$.

Insbesondere gilt:

$y1 \leq y2 \wedge \text{sorted } ys$.

Wir schließen:

```
... = sorted (y1 :: merge x (y2::ys'))
    = sorted (y1 :: y2 :: merge x ys')
    = sorted (y2 :: merge x ys')
    = sorted (merge x ys)
    = true nach Induktionsannahme
```

363

Fall 4.3: $ys = y2::ys' \wedge x1 \leq y2$.

Insbesondere gilt: $y1 < x1 \leq y2 \wedge \text{sorted } ys$.

Wir schließen:

```
... = sorted (y1 :: merge (x1::xs) (y2::ys'))
    = sorted (y1 :: x1 :: merge xs ys)
    = sorted (x1 :: merge xs ys)
    = sorted (merge x ys)
    = true nach Induktionsannahme
```

364

8 Parallele Programmierung

Die Bibliothek `threads.cma` unterstützt die Implementierung von Systemen, die mehr als einen Thread benötigen ...

Beispiel

```
module Echo = struct open Thread
  let echo () = print_string (read_line () ^ "\n")
  let main    = let t1 = create echo ()
                in join t1
                print_int (id (self ()));
                print_string "\n"
end
```

366

Diskussion

- Wieder steht der Beweis unter dem Vorbehalt, dass alle Aufrufe der Funktionen `sorted` und `merge` terminieren.
- Als zusätzliche Technik benötigten wir **Fallunterscheidungen** über die verschiedenen Möglichkeiten für Argumente in den Aufrufen.
- Die Fallunterscheidungen machten den Beweis länglich.
// Der Fall $n = 0$ ist tatsächlich überflüssig,
// da er in den Fällen 1 und 2 enthalten ist

365

Kommentar

- Die Struktur `Thread` versammelt Grundfunktionalität zur Erzeugung von Nebenläufigkeit.
- Die Funktion `create: ('a -> 'b) -> 'a -> t` erzeugt einen neuen Thread mit den folgenden Eigenschaften:
 - der Thread wertet die Funktion auf dem Argument aus;
 - der erzeugende Thread erhält die Thread-Id zurück und läuft unabhängig weiter.
 - Mit den Funktionen: `self : unit -> t` bzw. `id : t -> int` kann man die eigene Thread-Id abfragen bzw. in ein `int` umwandeln.

367

Weitere nützliche Funktionen

- Die Funktion `join: t -> unit` hält den aktuellen Thread an, bis die Berechnung des gegebenen Threads beendet ist.
- Die Funktion: `kill: t -> unit` beendet einen Thread;
- Die Funktion: `delay: float -> unit` verzögert den aktuellen Thread um eine Zeit in Sekunden;
- Die Funktion: `exit: unit -> unit` beendet den aktuellen Thread.

368

```
> ./a.out
> abcdefghijk
> abcdefghijk
> 0
>
```

- **Ocaml**-Threads werden vom System nur simuliert.
- Die Erzeugung von Threads ist **billig**.
- Die Programm-Ausführung endet mit der Terminierung des Threads mit der Id `0`.

370

Achtung

- In der interaktiven Umgebung können Threads mit Hilfe der Option `#thread;;` aktiviert werden.
- Stattdessen kann man mit der Option: `-thread` compilieren:
`> ocamlc -thread unix.cma threads.cma Echo.ml`
- Die Bibliothek `threads.cma` benötigt dabei Hilfsfunktionalität der Bibliothek `unix.cma`.
`//` unter Windows sieht die Sache vermutlich anders aus.
- Das Programm testen können wir dann durch Aufruf von:
`> ./a.out`

369

8.1 Kanäle

Threads kommunizieren über Kanäle.

Für Erzeugung, Senden auf und Empfangen aus einem Kanal stellt die Struktur `Event` die folgende Grundfunktionalität bereit:

```
type 'a channel
new_channel : unit -> 'a channel
type 'a event
always : 'a -> 'a event
sync : 'a event -> 'a
send : 'a channel -> 'a -> unit event
receive : 'a channel -> 'a event
```

371

- Jeder Aufruf `new_channel()` erzeugt einen anderen Kanal.
- Über einen Kanal können beliebige Daten geschickt werden !!!
- `always` wandelt einen Wert in ein Ereignis um.
- Senden und Empfangen erzeugen Ereignisse ...
- Synchronisierung auf Ereignisse liefert deren Wert.

```

module Exchange = struct open Thread open Event
let thread ch = let x = sync (receive ch)
                in print_string (x ^ "\n");
                sync (send ch "got it!")
let main = let ch = new_channel () in create thread ch;
           print_string "main is running ... \n";
           sync (send ch "Greetings!");
           print_string ("He " ^ sync (receive ch) ^ "\n")
end

```

372

Diskussion

- `sync (send ch str)` macht das Ereignis des Sendens der Welt offenbar und blockiert den Sender, bis jemand den Wert aus dem Kanal ausgelesen hat ...
- `sync (receive ch)` blockiert den Empfänger, bis ein Wert im Kanal enthalten ist. Dann liefert der Ausdruck diesen Wert.
- Synchrone Kommunikation ist eine Alternative zum Austausch von Daten zwischen Threads bzw. zur Organisation von Nebenläufigkeit
 \implies Rendezvous
- Insbesondere kann sie benutzt werden, um asynchrone Thread-Kooperation zu implementieren.

373

Im Beispiel spaltet `main` einen Thread ab. Dann sendet sie diesem einen String und wartet auf Antwort. Entsprechend wartet der Thread auf Übertragung eines `string`-Werts auf dem Kanal. Sobald er ihn erhalten hat, sendet er auf dem selben Kanal eine Antwort.

Achtung

Ist die Abfolge von `send` und `receive` nicht sorgfältig designt, können Threads leicht blockiert werden ...

Die Ausführung des Programms liefert:

```

> ./a.out
main is sending ...Greetings!
He got it!
>

```

374

Im Beispiel spaltet `main` einen Thread ab. Dann sendet sie diesem einen String und wartet auf Antwort. Entsprechend wartet der Thread auf Übertragung eines `string`-Werts auf dem Kanal. Sobald er ihn erhalten hat, sendet er auf dem selben Kanal eine Antwort.

Achtung

Ist die Abfolge von `send` und `receive` nicht sorgfältig designt, können Threads leicht blockiert werden ...

Die Ausführung des Programms liefert:

```

> ./a.out
main is sending ...Greetings!
He got it!
>

```

374

Beispiel: Eine globale Speicherzelle

Eine globale Speicherzelle, insbesondere in Anwesenheit mehrerer Threads sollte die Signatur `Cell` implementieren:

```
module type Cell = sig
  type 'a cell
  val new_cell : 'a -> 'a cell
  val get : 'a cell -> 'a
  val put : 'a cell -> 'a -> unit
end
```

Dabei muss sichergestellt werden, die verschiedenen `get`- und `put`-Aufrufe sequenzialisiert ausgeführt werden.

375

Diese Aufgabe erfüllt ein `Server`-Thread, mit dem `get` und `put` kommunizieren:

```
type 'a req = Get of 'a channel | Put of 'a
type 'a cell = 'a req channel
```

Der Kanal transportiert Requests an die Speicherzelle, welche entweder den zu setzenden Wert oder den Rückkanal enthalten ...

376