Script generated by TTT

Title: Seidl: Info2 (27.01.2017)

Date: Fri Jan 27 08:27:25 CET 2017

Duration: 87:43 min

Pages: 34

Allgemeiner

`app (rev x) y = 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
```


$n > 0$: 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
```


Allgemeiner

`app (rev x) y = 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
```

$n > 0$: 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
```


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:

  `rev x = rev1 x []`

  folgt aus:

  `app (rev x) y = rev1 x y`

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

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
```


Behauptung

`sorted x` $\wedge$ `sorted y` $\rightarrow$ `sorted (merge x y)`

für alle Listen `x, y`.

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

Gelte `sorted x` $\wedge$ `sorted y`.

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

Wir schließen:

```
sorted (merge x y)  =  sorted (merge [] [])
                    =  sorted []
                    =  true
```

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
```


Behauptung

sorted x ∧ sorted y → sorted (merge x y)
für alle Listen x, y.

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

Gelte sorted x ∧ sorted y.

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

Wir schließen:

```
sorted (merge x y)  =  sorted (merge [] [])
                    =  sorted []
                    =  true
```


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
```

359

Behauptung

sorted x ∧ sorted y → sorted (merge x y)
für alle Listen x, y.

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

Gelte sorted x ∧ sorted y.

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

Wir schließen:

```
sorted (merge x y)  =  sorted (merge [] [])
                    =  sorted []
                    =  true
```

360

$n > 0$:

Fall 1: `x = []`.

Wir schließen:

```
sorted (merge x y)  =  sorted (merge [] y)
                    =  sorted y
                    =  true
```

Fall 2: `y = []` analog.

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
```


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

Insbesondere gilt: `x1` $\leq$ `x2` $\wedge$ `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
```


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

Insbesondere gilt: `x1` $\leq$ `x2` $\wedge$ `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
```

Fall 3.3: `xs = x2::xs'` $\wedge$ `x2` $>$ `y1`.

Insbesondere gilt: `x1` $\leq$ `y1` $<$ `x2` $\wedge$ `sorted xs`.

Wir schließen:

```
...  =  sorted (x1 :: merge (x2::xs') (y1::ys))
     =  sorted (x1 :: y1 :: merge xs ys)
     =  sorted (y1 :: merge xs ys)
     =  sorted (merge xs y)
     =  true   nach Induktionsannahme
```


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
```


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

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
```

```
> ./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` .

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.

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.

Achtung

- Die interaktive Umgebung funktioniert nicht mit Threads !!
- Stattdessen muss 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`

```
> ./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` .

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
```


- 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
```


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!
>
```

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
  $\Longrightarrow$ Rendezvous
- Insbesondere kann sie benutzt werden, um asynchrone Thread-Kooperation zu implementieren.

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!
>
```


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 sequentialisiert ausgeführt werden.

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 ...

```
let get req = let reply = new_channel ()
              in  sync (send req (Get reply));
                  sync (receive reply)
```

Die Funktion `get` sendet einen neuen Rückkanal auf `req`. Ist dieser angekommen, wartet sie auf die Antwort.

```
let put req x = sync (send req (Put x))
```

Die Funktion `put` sendet ein `Put`-Element, das den neuen Wert der Speicherzelle enthält.


```
let get req = let reply = new_channel ()
              in  sync (send req (Get reply));
                  sync (receive reply)
```

Die Funktion `get` sendet einen neuen Rückkanal auf `req`. Ist dieser angekommen, wartet sie auf die Antwort.

```
let put req x = sync (send req (Put x))
```

Die Funktion `put` sendet ein `Put`-Element, das den neuen Wert der Speicherzelle enthält.

Spannend ist jetzt nur noch die Implementierung der Zelle selbst:

```
let new_cell x = let req = new_channel ()
          in let rec serve x = match sync (receive req)
             with Get reply -> sync (send reply x);
                               serve x
                | Put y     -> serve y
          in
             create serve x;
             req
```