Script generated by TTT

Title: Info2 (11.12.2015)

Date: Fri Dec 11 08:34:32 CET 2015

Duration: 82:25 min

Pages: 35

3.7 Anwendung: Queues

Gesucht:

Datenstruktur `'a queue`, die die folgenden Operationen unterstützt:

```
enqueue : 'a -> 'a queue -> 'a queue
dequeue : 'a queue  -> 'a option * 'a queue
is_empty : 'a queue -> bool
queue_of_list : 'a list -> 'a queue
list_of_queue : 'a queue -> 'a list
```


1. Idee

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial.

1. Idee

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial.

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function
      []    -> (None, [])
    | x::xs -> (Some x, xs)
```

1. Idee

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

  Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial.
- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function
      []    -> (None, [])
    | x::xs -> (Some x, xs)
```

- Einfügen bedeutet hinten anhängen:

```
let enqueue x xs = xs @ [x]
```


Diskussion

- Der Operator `@` konkateniert zwei Listen.
- Die Implementierung ist sehr einfach.
- Entnehmen ist sehr billig.
- Einfügen dagegen kostet so viele rekursive Aufrufe von `@` wie die Schlange lang ist.
- Geht das nicht besser ??

2. Idee

- Repräsentiere die Schlange als zwei Listen !!!

```
type 'a queue = Queue of 'a list * 'a list
let is_empty = function
               Queue ([],[]) -> true
             | _             -> false
let queue_of_list list = Queue (list,[])
let list_of_queue = function
               Queue (first,[])   -> first
             | Queue (first,last) ->
                      first @ List.rev last
```

- Die zweite Liste repräsentiert das Ende der Liste und ist deshalb in umgedrehter Anordnung ...

2. Idee (Fortsetzung)

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =
               Queue (first, x::last)
```

2. Idee (Fortsetzung)

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =
                Queue (first, x::last)
```

- Entnahme bezieht sich dagegen auf die erste Liste.

  Ist diese aber leer, wird auf die zweite zugegriffen ...

```
let dequeue = function
    Queue ([],last) -> (match List.rev last
                with [] -> (None, Queue ([],[]))
                | x::xs -> (Some x, Queue (xs,[])))
  | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```


Diskussion

- Jetzt ist Einfügen billig!
- Entnehmen dagegen kann so teuer sein, wie die Anzahl der Elemente in der zweiten Liste ...
- Gerechnet aber auf jede Einfügung, fallen nur konstante Zusatzkosten an !!!

$\Longrightarrow$ amortisierte Kostenanalyse

3.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen Daten. Daten, z.B. `[1;2;3]` können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- `fun` leitet eine Abstraktion ein.
  Der Name kommt aus dem λ-Kalkül.
- `->` hat die Funktion von `=` in Funktionsdefinitionen.
- Rekursive Funktionen können so nicht definiert werden, denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen.

Namenlose Funktionen werden verwendet, wenn sie nur einmal im Programm vorkommen. Oft sind sie Argument für Funktionale:

```
# map (fun x -> x*x) [1;2;3];;
- : int list = [1; 4; 9]
```

Oft werden sie auch benutzt, um eine Funktion als Ergebnis zurückzuliefern:

```
# let make_undefined () = fun x -> None;;
val make_undefined : unit -> 'a -> 'b option = <fun>
# let def_one (x,y) = fun x' -> if x=x' then Some y
                      else None;;
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

- Um Pattern Matching zu benutzen, kann man `match ... with` für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich `function` an ...

```
# function None   -> 0
          | Some x -> x*x+1;;
- : int option -> int = <fun>
```


- Um Pattern Matching zu benutzen, kann man `match ... with` für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich `function` an ...

```
# function None   -> 0
          | Some x -> x*x+1;;
- : int option -> int = <fun>
```

Alonzo Church, 1903–1995

3.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen Daten. Daten, z.B. `[1;2;3]` können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- `fun` leitet eine Abstraktion ein.
  Der Name kommt aus dem λ-Kalkül.
- `->` hat die Funktion von `=` in Funktionsdefinitionen.
- Rekursive Funktionen können so nicht definiert werden, denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen.

4 Größere Anwendung: Balancierte Bäume

Erinnerung: Sortiertes Array

5 Praktische Features in Ocaml

- Ausnahmen
- Ein- und Ausgabe als Seiteneffekte
- Sequenzen

5.1 Ausnahmen (Exceptions)

Bei einem Laufzeit-Fehler, z.B. Division durch Null, erzeugt das Ocaml-System eine exception (Ausnahme):

```
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".
```

Hier werden die Ausnahmen `Division_by_zero`, `Failure "tl"` bzw. `Invalid_argument "Char.chr"` erzeugt.

Ein anderer Grund für eine Ausnahme ist ein unvollständiger Match:

```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
```

In diesem Fall wird die Exception `Match_failure ("", 2, -9)` erzeugt.

Vordefinierte Konstruktoren für Exceptions

`Division_by_zero`	Division durch Null
`Invalid_argument of string`	falsche Benutzung
`Failure of string`	allgemeiner Fehler
`Match_failure of string * int * int`	unvollständiger Match
`Not_found`	nicht gefunden
`Out_of_memory`	Speicher voll
`End_of_file`	Datei zu Ende
`Exit`	für die Benutzerin ...

Eine Exception ist ein First Class Citizen, d.h. ein Wert eines Datentyps `exn` ...

Vordefinierte Konstruktoren für Exceptions

`Division_by_zero`	Division durch Null
`Invalid_argument of string`	falsche Benutzung
`Failure of string`	allgemeiner Fehler
`Match_failure of string * int * int`	unvollständiger Match
`Not_found`	nicht gefunden
`Out_of_memory`	Speicher voll
`End_of_file`	Datei zu Ende
`Exit`	für die Benutzerin ...

Eine Exception ist ein First Class Citizen, d.h. ein Wert eines Datentyps `exn` ...


```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` erweitert wird ...

```
# exception Hell of string;;
exception Hell of string
# Hell "damn!";;
- : exn = Hell "damn!"
```



```
let rec member x l = try if x = List.hd l then true
                         else member x (List.tl l)
    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Das Schlüsselwort `with` leitet ein Pattern Matching auf dem Ausnahme-Datentyp `exn` ein:

```
try <exp>
with <pat1> -> <exp1> | ...  | <patN> -> <expN>
```

$\Longrightarrow$ Man kann mehrere Exceptions gleichzeitig abfangen

Ausnahmebehandlung

Wie in Java können Exceptions ausgelöst und behandelt werden:

```
# let teile (n,m) = try Some (n / m)
      with Division_by_zero -> None;;

# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

So kann man z.B. die member-Funktion neu definieren:


```
let rec member x l = try if x = List.hd l then true
                         else member x (List.tl l)
    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Das Schlüsselwort `with` leitet ein Pattern Matching auf dem Ausnahme-Datentyp `exn` ein:

```
try <exp>
with <pat1> -> <exp1> | ...  | <patN> -> <expN>
```

$\Longrightarrow$ Man kann mehrere Exceptions gleichzeitig abfangen

Der Programmierer kann selbst Exceptions auslösen.
Das geht mit dem Schlüsselwort `raise` ...

```
# 1 + (2/0);;
Exception: Division_by_zero.
# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception.

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                        with Division_by_zero ->
                             raise (Failure "Division by zero")
                    in string_of_int (n*n)
              with Failure str -> "Error: " ^ str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

5.2 Textuelle Ein- und Ausgabe

- Lesen aus der Eingabe und Schreiben auf die Ausgabe sprengt den rein funktionalen Rahmen !
- Diese Operationen werden darum mit Hilfe von Seiteneffekten realisiert, d.h. mit Hilfe von Funktionen, deren Rückgabewert uninteressant ist (etwa `unit`).
- Während der Ausführung wird dann aber die entsprechende Aktion ausgeführt

  $\Longrightarrow$ nun kommt es genau auf die Reihenfolge der Auswertung an !!!

- Selbstverständlich kann man in Ocaml auf den Standard-Output schreiben:

```
# print_string "Hello World!\n";;
Hello World!
- : unit = ()
```

- Analog gibt es eine Funktion: `read_line : unit -> string` ...

```
# read_line ();;
Hello World!
- : "Hello World!"
```


Um aus einer Datei zu lesen, muss man diese zum Lesen öffnen ...

```
# let infile = open_in "test";;
val infile : in_channel = <abstr>
# input_line infile;;
- : "Die einzige Zeile der Datei ...";;
# input_line infile;;
Exception: End_of_file
```

Gibt es keine weitere Zeile, wird die Exception `End_of_file` geworfen.

Benötigt man einen Kanal nicht mehr, sollte man ihn geregelt schließen ...

```
# close_in infile;;
- : unit = ()
```


Weitere nützliche Funktionen

```
stdin              : in_channel
input_char         : in_channel -> char
in_channel_length : in_channel -> int
input : in_channel -> string -> int -> int -> int
```

- `in_channel_length` liefert die Gesamtlänge der Datei.
- `input chan buf p n` liest aus einem Kanal `chan` `n` Zeichen und schreibt sie ab Position `p` in den String `buf`.

Weitere nützliche Funktionen

```
stdin             : in_channel
input_char        : in_channel -> char
in_channel_length : in_channel -> int
input : in_channel -> string -> int -> int -> int
```

- `in_channel_length` liefert die Gesamtlänge der Datei.
- `input chan buf p n` liest aus einem Kanal `chan` `n` Zeichen und schreibt sie ab Position `p` in den String `buf`.

Die Ausgabe in Dateien erfolgt ganz analog ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelt geschlossen wurde ...

```
# close_out outfile;;
- : unit = ()
```


5.3 Sequenzen

Bei Seiteneffekten kommt es auf die Reihenfolge an!

Mehrere solche Aktionen kann man mit dem Sequenz-Operator `;` hintereinander ausführen:

```
# print_string "Hello";
  print_string " ";
  print_string "world!\n";;
Hello world!
- : unit = ()
```


Oft möchte man viele Strings ausgeben !

Hat man etwa eine Liste von Strings, hilft das Listenfunktional `List.iter`: weiter:

```
#  let rec iter f = function
       []    -> ()
     | x::[] -> f x
     | x::xs -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>
```