Script generated by TTT

Title: Seidl: Info2 (23.12.2016)

Date: Fri Dec 23 08:28:54 CET 2016

Duration: 87:00 min

Pages: 46

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 !!!

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!"
```


- 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 = ()
```


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>
```


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>
```


6 Das Modulsystem von OCAML

- → Strukturen
- → Signaturen
- → Information Hiding
- → Funktoren
- → Getrennte Übersetzung

6.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet Ocaml Module oder Strukturen an:

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb nicht sichtbar:

```
# first;;
Unbound value first
```


Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. mehrere Funktionen gleichen Namens definieren:

```
# module Triples = struct
    type 'a triple = Triple of 'a * 'a * 'a
    let first (Triple (a,_,_)) = a
    let second (Triple (_,b,_)) = b
    let third (Triple (_,_,c)) = c
  end;;
...
```



```
...
module Triples :
sig
  type 'a triple = Triple of 'a * 'a * 'a
  val first : 'a triple -> 'a
  val second : 'a triple -> 'a
  val third : 'a triple -> 'a
end
# Triples.first;;
- : 'a Triples.triple -> 'a = <fun>
```


... oder mehrere Implementierungen der gleichen Funktion:

```
# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```

... oder mehrere Implementierungen der gleichen Funktion:

```
# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```


Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb nicht sichtbar:

```
# first;;
Unbound value first
```


Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man alle Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls Bestandteil des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...


```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```

Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man alle Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls Bestandteil des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...


```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```


Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```



```
    ...
    let first q = Pairs.first (Pairs.first q)
    let second q = Pairs.second (Pairs.first q)
    let third q = Pairs.first (Pairs.second q)
    let fourth q = Pairs.second (Pairs.second q)
  end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

6.2 Modul-Typen oder Signaturen

Mithilfe von Signaturen kann man einschränken, was eine Struktur nach außen exportiert.

Explizite Angabe einer Signatur gestattet:

- die Menge der exportierten Variablen einzuschränken;
- die Menge der exportierten Typen einzuschränken ...

... ein Beispiel:


```
module Sort = struct
    let single list = map (fun x->[x]) list
    let rec merge l1 l2 = match (l1,l2)
        with ([],_) -> l2
        |    (_,[]) -> l1
        | (x::xs,y::ys) -> if x<y then x :: merge xs l2
                                  else y :: merge l1 ys
    let rec merge_lists = function
            [] -> [] | [l] -> [l]
    | l1::l2::ll -> merge l1 l2 :: merge_lists ll
    let sort list = let list = single list
        in let rec doit = function
           [] -> [] | [l] -> l
        |   l -> doit (merge_lists l)
    in doit list
end
```


Die Implementierung macht auch die Hilfsfunktionen `single`, `merge` und `merge_lists` von außen zugreifbar:

```
# Sort.single [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```

Damit die Funktionen `single` und `merge_lists` nicht mehr exportiert werden, verwenden wir die Signatur:

```
module type Sort = sig
    val merge : 'a list -> 'a list -> 'a list
    val sort : 'a list -> 'a list
  end
```


Die Funktionen `single` und `merge_lists` werden nun nicht mehr exportiert.

```
# module MySort : Sort = Sort;;
module MySort : Sort
# MySort.single;;
Unbound value MySort.single
```

```
module Sort = struct
    let single list = map (fun x->[x]) list
    let rec merge l1 l2 = match (l1,l2)
        with ([],_) -> l2
        |    (_,[]) -> l1
        | (x::xs,y::ys) -> if x<y then x :: merge xs l2
                                  else y :: merge l1 ys
    let rec merge_lists = function
            [] -> [] | [l] -> [l]
    | l1::l2::ll -> merge l1 l2 :: merge_lists ll
    let sort list = let list = single list
        in let rec doit = function
           [] -> [] | [l] -> l
        |   l -> doit (merge_lists l)
    in doit list
end
```


Die Funktionen `single` und `merge_lists` werden nun nicht mehr exportiert.

```
# module MySort : Sort = Sort;;
module MySort : Sort
# MySort.single;;
Unbound value MySort.single
```


Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein.

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```


Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein.

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```

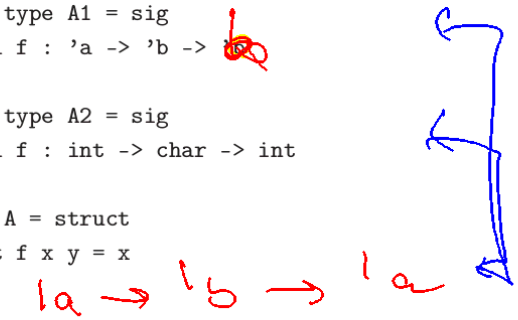
299

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein.

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'a
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```




```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                                  is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```


6.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _  -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```


Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs queue zu verschleiern.

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```



```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs queue zu verschleiern.

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs queue zu verschleiern.

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```


6.4 Funktoren

Da in Ocaml fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordung gibt: die Funktoren.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.