Cheat sheet FPV OCaml

Funktionale Programmierung (Technische Universität München)

Allgemeine Begriffe

- **Currying:** Umwandlung einer Funktion mit mehreren Argumenten in eine Funktion mit einem Argument:
- **Funktion höherer Ordnung:** haben Funktionen als Argumente und/oder liefern Funktionen als Ergebnisse.

```
let plus2 = (+) 2
val plus2 : int -> int = <fun>
```

- **Strict evaluation:** Ausdruuecke werden sofort ausgewertet.
- **Lazy evaluation:** Ausdruuecke werden ausgewertet, sobald ihr Wert benoetigt wird.
- **Local Consistency:** An arbitrary pre-condition A for a statement s is valid whenever $A \Rightarrow \mathbf{WP}[\![s]\!](B)$.
  Gilt diese Implikation so ist es locally consistent
- **Polymorphic Functions:** Funktion mit polymorphen parametern ('a) oder Datentypen, die polymorphic sind, sind polymorphic

OCaml

Basics

Record:

```
type pair = {a: int; b: int }
```

Match mit:

```
... with | {a=x; b=y} -> z.B. x + y
```

Oder (name der vars muss gleicher name wie in record definition, reihenfolge ist egal)

```
... with | {a; b} -> z.B. a + b
```

Man kann auch nur auf teile des tupels matchen, der rest ist dann egal (name der vars muss gleiche reihenfolge und gleicher name wie in record definition)

```
... with | {a} -> z.B. a * a
... with | {a=value; b=_} -> z.B. a * a
```

Tuple:

```
type iil = int * int list
```

Variant(Konstruktor, Enumeration Type):

```
type foo = Nothing | Int of int | Pair of int *
↪  int
```

Mit parameter

```
type 'a my_list = Empty | Cons of 'a * 'a my_list
```

Mehrere Parameter:

```
type ('a, 'b) my_list = Empty | Cons of ('a * 'b)
↪  * ('a, 'b) my_list
val r : (int, int) my_list = Cons ((1, 2), Empty)
```

Modules

Signature:

```
module type Set = sig
  type t
  val to_string : t -> string
end
```

Signature Extension:

```
module type OrderedSet = sig
    include Set
    val compare : t -> t -> int
end
```

Implementation:

```
module StringSet : Set
    with type t = string = struct
    type t = string
    let to_string s = "\"" ^ s ^ "\""
end
```

Functors

Definition:

```
module BTreeMap
    (KeySet : OrderedSet)
    (ValueSet : Set) : Map
    with type key = KeySet.t
    and type value = ValueSet.t
    = struct
    type key = KeySet.t
    type value = ValueSet.t
    type t = Empty | Node of (key * value) * t * t
end
```

Modul List

```
val find_opt :
    ('a -> bool) -> 'a list -> 'a option
```

`find_opt p l` returns the first element of the list l that satisfies the predicate p, or None if there is no value that satisfies p in the list l.

```
val split : ('a * 'b) list -> 'a list * 'b list
```

Transform a list of pairs into a pair of lists: `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`. Not tail-recursive.

```
val combine :
    'a list -> 'b list -> ('a * 'b) list
```

Transform a pair of lists into a list of pairs: `combine [a1; ...; an] [b1; ...; bn]` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

```
val init : int -> (int -> 'a) -> 'a list
```

List.init len f is f 0; f 1; ...; f (len-1), evaluated left to right.

```
val nth : 'a list -> int -> 'a
val rev : 'a list -> 'a list
val iter : ('a -> unit) -> 'a list -> unit
val iteri : (int -> 'a -> unit) -> 'a list -> unit
(* for iteri funciton also gets index *)
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b
↪  list -> unit
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list
↪  -> 'c list
```

Exceptions

General Error:

```
raise Failure "some string"
```

Define Exceptions:

```
exception Hell of string
raise (Hell "damn!")
```

Exception Handling:

```
try <expr>
 ith <pat1> -> <exp1> | ... | <patN> -> <expN>
```

I/O:

```
(* out_channel *)
let file = open_out filename in
(* unit *)
let _ = Printf.fprintf file "%d;%.2f\n" c g in
(* in_channel *)
let file = open_in filename
(* string *)
try let line = input_line file ...
with End_of_file -> ...
(* Closing! *)
close_in file (* -> () *)
(* normaler print auf Commandline *)
print_string "HW\n"
(* input von der Commandline*)
read_line ();;
```

Write Beispiel:

```
let file = open_out filename in
  let rec write t = Printf.fprintf file "%s\n" t
  in
  List.iter write strings;
  close_out file
```

Read Beispiel(tail recursive):

```
let lines p = let ch = open_in p
    in let rec h xs = try h (input_line ch::xs)
    ↪  with
       End_of_file -> close_in ch; rev xs
    in h []
```

Concurrency

Threads Basics:

```
(* returns handle to newly created thread *)
let th = Thread.create "<function>" "<value>" in
Thread.join th
```

Memory Cell:

```
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

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

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

let put cell x = sync (send cell (Put x))

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

Lazy Evaluation

Definition:

```
type 'a llist = Cons of 'a * (unit -> 'a llist)
```

Implementation:

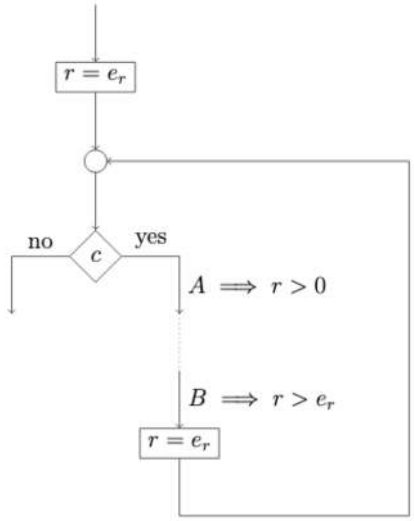
```
let rec lnat i = Cons (i, (fun () -> lnat (i +
↪  1)))

let lfib () =
    let rec impl a b =
        Cons (a, fun () -> impl b (a+b))
    in
    impl 0 1

let rec ltake n (Cons (h, t)) =
    if n <= 0 then
    []
    else h::ltake (n-1) (t ())

let rec lfilter f (Cons (h, t)) =
    if f h then
    Cons (h, fun () -> lfilter f (t ()))
    else lfilter f (t ())
```

Verification

Weakest Preconditions

- $\mathbf{WP}[\![\texttt{;}]\!](B) \equiv B$
- $\mathbf{WP}[\![\texttt{x = e;}]\!](B) \equiv B[e/x]$
- $\mathbf{WP}[\![\texttt{x = read();}]\!](B) \equiv \forall x.B$
- $\mathbf{WP}[\![\texttt{write(x);}]\!](B) \equiv B$
- $\mathbf{WP}[\![\texttt{condition b}]\!](B_0, B_1) \equiv (\neg b \wedge B_0) \vee (b \wedge B_1)$

Termination



I (vor c) ist meist r = ... und noch zusätzliche Bedingungen. Von I nach B faellt der teil mit r weg, aber mann kann es staerker machen, indem man es einfach mit $r > e_r$ wieder hinzufügt

Mathematik

$$i + 1 \leq n \Rightarrow i < n$$

$$i - 1 \geq n \Rightarrow i > n$$