Script generated by TTT

Title: Info2 (04.12.2015)

Date: Fri Dec 04 08:36:03 CET 2015

Duration: 87:49 min

Pages: 25

Das Prinzip heißt nach seinem Erfinder Haskell B. Curry Currying.

→ g heißt Funktion höherer Ordnung, weil ihr Ergebnis wieder eine Funktion ist.

→ Die Anwendung von g auf ein Argument heißt auch partiell, weil das Ergebnis nicht vollständig ausgewertet ist, sondern eine weitere Eingabe erfordert.

Das Argument einer Funktion kann auch wieder selbst eine Funktion sein:

```
# let apply f a b = f (a,b);;
val apply ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
```


Das Prinzip heißt nach seinem Erfinder Haskell B. Curry Currying.

→ g heißt Funktion höherer Ordnung, weil ihr Ergebnis wieder eine Funktion ist.

→ Die Anwendung von g auf ein Argument heißt auch partiell, weil das Ergebnis nicht vollständig ausgewertet ist, sondern eine weitere Eingabe erfordert.

Das Argument einer Funktion kann auch wieder selbst eine Funktion sein:

```
# let apply f a b = f (a,b);;
val apply ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
```



```
...
# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# apply plus;;
- : int -> int -> int = <fun>
# let plus2 = apply plus 2;;
val plus2 : int -> int = <fun>
# let plus3 = apply plus 3;;
val plus3 : int -> int = <fun>
# plus2 (plus3 4);;
- : int = 9
```

3.3 Funktionen als Daten

Funktionen sind Daten und können daher in Datenstrukturen vorkommen:

```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
              []    -> []
            | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]
```

// (+) : int -> int -> int ist die Funktion zum Operator +


```
...
# let do_add n =
      let rec add_list = function
                  [] -> []
              | f::fs -> f n :: add_list fs
  in add_list ;;
val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
# do_add 5 l;;
- : int list = [6;7;8]

# let rec sum = function
            [] -> 0
      | f::fs -> f (sum fs);;
val sum : (int -> int) list -> int = <fun>
# sum l;;
- : int = 6
```


3.3 Funktionen als Daten

Funktionen sind Daten und können daher in Datenstrukturen vorkommen:

```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
              []    -> []
            | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]
```

// (+) : int -> int -> int ist die Funktion zum Operator +

196

```
...
# let do_add n =
      let rec add_list = function
                  [] -> []
              | f::fs -> f n :: add_list fs
  in add_list ;;
val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
# do_add 5 l;;
- : int list = [6;7;8]

# let rec sum = function
            [] -> 0
      | f::fs -> f (sum fs);;
val sum : (int -> int) list -> int = <fun>
# sum l;;
- : int = 6
```

197

3.3 Funktionen als Daten

Funktionen sind Daten und können daher in Datenstrukturen vorkommen:

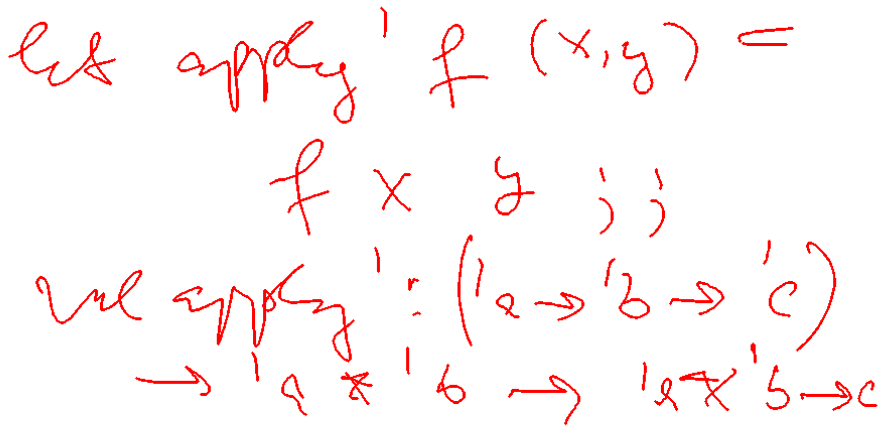
```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
              []    -> []
            | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]
```

// `(+) : int -> int -> int` ist die Funktion zum Operator `+`


```
...
# let do_add n =
      let rec add_list = function
                  [] -> []
                | f::fs -> f n :: add_list fs
  in add_list ;;
val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
# do_add 5 l;;
- : int list = [6;7;8]

# let rec sum = function
            [] -> 0
      | f::fs -> f (sum fs);;
val sum : (int -> int) list -> int = <fun>
# sum l;;
- : int = 6
```


3.4 Einige Listen-Funktionen

```
let rec map f = function
            [] -> []
      | x::xs -> f x :: map f xs

let rec  fold_left f a = function
            [] -> a
      | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
            [] -> fun b -> b
      | x::xs -> fun b -> f x (fold_right f xs b)
```

3.4 Einige Listen-Funktionen

```
let rec map f = function
          [] -> []
      | x::xs -> f x :: map f xs

let rec  fold_left f a = function
          [] -> a
      | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
          [] -> fun b -> b
      | x::xs -> fun b -> f x (fold_right f xs b)
```



```
let rec  find_opt f = function
          [] -> None
      | x::xs -> if f x then Some x
                 else find_opt f xs
```

Beachte

→ Diese Funktionen abstrahieren von dem Verhalten der Funktion `f`. Sie spezifizieren das Rekursionsverhalten gemäß der Listenstruktur, unabhängig von den Elementen der Liste.

→ Daher heißen solche Funktionen Rekursions-Schemata oder (Listen-)Funktionale.

→ Listen-Funktionale sind unabhängig vom Typ der Listenelemente. (Diesen muss nur die Funktion `f` kennen.

→ Funktionen, die gleich strukturierte Eingaben verschiedenen Typs verarbeiten können, heißen polymorph.

3.4 Einige Listen-Funktionen

```
let rec map f = function
          [] -> []
      | x::xs -> f x :: map f xs

let rec  fold_left f a = function
          [] -> a
      | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
          [] -> fun b -> b
      | x::xs -> fun b -> f x (fold_right f xs b)
```

198

```
let rec  find_opt f = function
          [] -> None
      | x::xs -> if f x then Some x
                 else find_opt f xs
```

Beachte

→ Diese Funktionen abstrahieren von dem Verhalten der Funktion `f`. Sie spezifizieren das Rekursionsverhalten gemäß der Listenstruktur, unabhängig von den Elementen der Liste.

→ Daher heißen solche Funktionen Rekursions-Schemata oder (Listen-)Funktionale.

→ Listen-Funktionale sind unabhängig vom Typ der Listenelemente. (Diesen muss nur die Funktion `f` kennen.

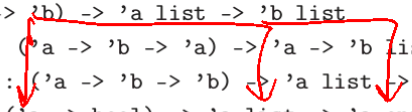
→ Funktionen, die gleich strukturierte Eingaben verschiedenen Typs verarbeiten können, heißen polymorph.

199

3.5 Polymorphe Funktionen

Das Ocaml-System inferiert folgende Typen für diese Funktionale:

```
map : ('a -> 'b) -> 'a list -> 'b list
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
find_opt : ('a -> bool) -> 'a list -> 'a option
```



→ 'a und 'b sind Typvariablen. Sie können durch jeden Typ ersetzt (instanziiert) werden (aber an jedem Vorkommen durch den gleichen Typ).

→ Durch partielle Anwendung auf eine Funktion können die Typvariablen instanziiert werden:

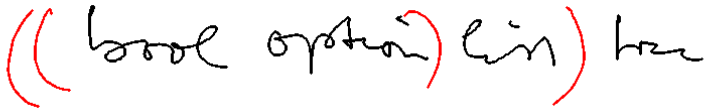
```
# Char.chr;;
val : int -> char = <fun>
# map Char.chr;;
- : int list -> char list = <fun>

# fold_left (+);;
val it : int -> int list -> int = <fun>
```


→ Wenn man einem Funktional eine polymorphe Funktion als Argument gibt, ist das Ergebnis wieder polymorph:

```
# let cons_r xs x = x::xs;;
val cons_r : 'a list -> 'a -> 'a list = <fun>
# let rev l = fold_left cons_r [] l;;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3];;
- : int list = [3; 2; 1]
# rev [true;false;false];;
- : bool list = [false; false; true]
```


Ein paar der einfachsten polymorphen Funktionen:

```
let compose f g x = f (g x)
let twice f x = f (f x)
let iter f g x = if g x then x else iter f g (f x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
val iter : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>

# compose neg neg;;
- : bool -> bool = <fun>
# compose neg neg true;;
- : bool = true;;
# compose Char.chr plus2 65;;
- : char = 'C'
```



3.6 Polymorphe Datentypen

Man kann sich auch selbst poloymorphe Datentypen definieren:

```
type 'a tree = Leaf of 'a
             | Node of ('a tree * 'a tree)
```

→ `tree` heißt Typkonstruktor, weil er aus einem anderen Typ (seinem Parameter `'a`) einen neuen Typ erzeugt.

→ Auf der rechten Seite dürfen nur die Typvariablen vorkommen, die auf der linken Seite als Argument für den Typkonstruktor stehen.

→ Die Anwendung der Konstruktoren auf Daten instanziiert die Typvariable(n):


```
# Leaf 1;;
- : int tree = Leaf 1
# Node (Leaf ('a',true), Leaf ('b',false));;
- : (char * bool) tree = Node (Leaf ('a', true),
                               Leaf ('b', false))
```

Funktionen auf polymorphen Datentypen sind typischerweise wieder polymorph ...


```
let rec  size = function
       Leaf _     -> 1
    |  Node(t,t') -> size t + size t'

let rec flatten = function
       Leaf x     -> [x]
     | Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit = function
                     (Leaf x, xs) -> x :: xs
                | (Node(t,t'), xs) -> let xs = doit (t',xs)
                                      in  doit (t,xs)
   in doit (t,[])
...
```


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
```

```
...
val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node(Node(Leaf 1,Leaf 5),Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)

# size t;;
- : int = 3
# flatten t;;
val : int list = [1;5;3]
# flatten1 t;;
val : int list = [1;5;3]
```


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
```