Script generated by TTT

Title: Seidl: Functional Programming and Verification (30.11.2018)

Date: Fri Nov 30 08:34:40 CET 2018

Duration: 85:59 min

Pages: 16

3.1 Last Calls

A last call in the body e of a function is a call whose value provides the value of e ...

```
let f x = x+5
let g y = let z = 7
          in if y>5 then f (-y)
             else z + f y
```

The first call is last, the second is not.

⟹ From a last call, we need not return to the calling function.

⟹ The stack space of the calling function can immediately be recycled !!!

3.2 Higher Order Functions

Consider the two functions

```
let f (a,b) = a+b+1;;
let g a b   = a+b+1;;
```

At first sight, f and g differ only in the syntax. But they also differ in their types:

```
# f;;
- : int * int -> int = <fun>
# g;;
- : int -> int -> int = <fun>
```


- Function f has a single argument, namely, the pair (a,b). The return value is given by a+b+1.
- Function g has the argument a of type int. The result of application to a is again a function that, when applied to another argument b, returns the result a+b+1 :

```
# f (3,5);;
- : int = 9
# let g1 = g 3;;
val g1 : int -> int = <fun>
# g1 5;;
- : int = 9
```

Haskell B. Curry, 1900–1982

In honor of its inventor Haskell B. Curry, this principle is called Currying.

→ g is called a higher order function, because its result is again a function.

→ The application of g to a single argument is called partial, because the result takes another argument, before the body is evaluated.

The argument of a function can again be a function:

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


Haskell B. Curry, 1900–1982

In honor of its inventor Haskell B. Curry, this principle is called Currying.

→ g is called a higher order function, because its result is again a function.

→ The application of g to a single argument is called partial, because the result takes another argument, before the body is evaluated.

The argument of a function can again be a function:

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

3.3 Some List Functions

```
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 map_tr f list =
    let rec lift a l = match l with
        | [] -> a
        | x::xs -> lift (f x :: a) xs
    in lift [] l
```

3.3 Some List Functions

```
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       map_tr f list =
    let rec lift a l = match l with
        | [] -> a
        | x::xs -> lift (f x :: a) xs
    in lift [] l
```

3.3 Some List Functions

```
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
```

Remarks

→ These functions abstract from the behavior of the function `f`. They specify the recursion according the list structure — independently of the elements of the list.

→ Therefore, such functions are sometimes called recursion schemes or (list) functionals.

→ List functionals are independent of the element type of the list. That type must only be known to the function `f`.

→ Functions which operate on equally structured data of various type, are called polymorphic.

3.4 Polymorphic Functions

The Ocaml system infers the following types for the given functionals:

```
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 and 'b are type variables. They can be instantiated by any type (but each occurrence with the same type).

→ If a functional is applied to a function that is itself polymorphic, the result may again be polymorphic:

```
# 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]
```

Some of the Simplest Polymorphic Functions

```
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'
```