Script generated by TTT

Title: Seidl: Functional Programming and Verification (23.11.2018)

Date: Fri Nov 23 08:29:55 CET 2018

Duration: 88:35 min

Pages: 14

Case distinction for several arguments

```
# let rec app l y = match l
                    with [] -> y
                    | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

... can also be written as

```
# let rec app = function [] -> fun y -> y
                       | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```


Example: Playing cards

2. Idea: Enumeration Types

```
# type color = Diamonds | Hearts | Gras | Clubs;;
type color = Diamonds | Hearts | Gras | Clubs
# type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace;;
type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace
# Clubs;;
- : color = Clubs
# let gras_jack = (Gras,Jack);;
val gras_jack : color * value = (Gras,Jack)
```


Example: Playing cards

2. Idea: Enumeration Types

```
# type color = Diamonds | Hearts | Gras | Clubs;;
type color = Diamonds | Hearts | Gras | Clubs
# type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace;;
type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace
# Clubs;;
- : color = Clubs
# let gras_jack = (Gras,Jack);;
val gras_jack : color * value = (Gras,Jack)
```

171

```
# let string_of_color = function
                        Diamonds  -> "Diamonds"
                      | Hearts  -> "Hearts"
                      | Gras    -> "Gras"
                      | Clubs -> "Clubs";;
val string_of_color : color -> string = <fun>
```

Remark

The function `string_of_color` returns for a given color the corresponding string in constant time (the compiler, hopefully, uses jump tables).

Now, Ocaml can (almost) play cards:

```
# let takes = function
            |  ((f1,Queen),(f2,Queen))    -> f1 > f2
            |  ((_,Queen),_)              -> true
            |  (_,(_,Queen))              -> false
            |  ((f1,Jack),(f2,Jack))   -> f1 > f2
            |  ((_,Jack),_)              -> true
            |  (_,(_,Jack))              -> false
            |  ((Hearts,w1),(Hearts,w2))    -> w1 > w2
            |  ((Hearts,_),_)              -> true
            |  (_,(Hearts,_))              -> false
            |  ((f1,w1),(f2,w2))         -> if f1=f2 then w1 > w2
                                            else false;;
```



```
  ...
# let take (card2,card1) =
      if takes (card2,card1) then card2 else card1;;

# let trick (card1,card2,card3,card4) =
      take (card4, take (card3, take (card2,card1)));;

# trick ((Gras,Ace),(Gras,Nine),(Hearts,Ten),(Clubs,Jack));;
- : color * value = (Clubs,Jack)
# trick ((Clubs,Eight),(Clubs,King),(Gras,Ten),
         (Clubs,Nine));;
- : color * value = (Clubs,King)
```


Sum Types

Sum types generalize of enumeration types in that constructors now may have arguments.

Example: Hexadecimal numbers

```
type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
       then (Char.code c)-55
    else if c >= 'a' && c <= 'f'
       then (Char.code c)-87
    else -1;;
let hex2dez = function
              Digit n -> n
            | Letter c -> char2dez c;;
```

Char is a module, which collects useful functions and values for char.

A constructor defined by `type t = Con of <type> | ...` has functionality `Con : <type> -> t` — must, however, always occur applied ...

```
# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10
```


Datatypes can be recursive:

```
type sequence = End | Next of (int * sequence)

# Next (1, Next (2, End));;
- : sequence = Next (1, Next (2, End))
```

Note the similarity to lists!

Sum Types

Sum types generalize of enumeration types in that constructors now may have arguments.

Example: Hexadecimal numbers

```
type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
       then (Char.code c)-55
    else if c >= 'a' && c <= 'f'
       then (Char.code c)-87
    else -1;;
let hex2dez = function
              Digit n -> n
            | Letter c -> char2dez c;;
```


Recursive datatypes lead to recursive functions:

```
# let rec nth = function
                  (_,End) -> -1
                | (0,Next (x,_)) -> x
                | (n,Next (_, rest)) -> nth (n-1,rest);;
val nth : int * sequence -> int = <fun>

# nth (4, Next (1, Next (2, End)));;
- : int = -1
# nth (2, Next (1, Next(2, Next (5, Next (17, End)))));;
- : int = 5
```

Now, Ocaml can (almost) play cards:

```
# let takes = function
            |  ((f1,Queen),(f2,Queen))    -> f1 > f2
            |  ((_,Queen),_)              -> true
            |  (_,(_,Queen))              -> false
            |  ((f1,Jack),(f2,Jack))  -> f1 > f2
            |  ((_,Jack),_)             -> true
            |  (_,(_,Jack))             -> false
            |  ((Hearts,w1),(Hearts,w2))    -> w1 > w2
            |  ((Hearts,_),_)             -> true
            |  (_,(Hearts,_))             -> false
            |  ((f1,w1),(f2,w2))        -> if f1=f2 then w1 > w2
                                           else false;;
```



```
  ...
# let take (card2,card1) =
      if takes (card2,card1) then card2 else card1;;

# let trick (card1,card2,card3,card4) =
      take (card4, take (card3, take (card2,card1)));;

# trick ((Gras,Ace),(Gras,Nine),(Hearts,Ten),(Clubs,Jack));;
- : color * value = (Clubs,Jack)
# trick ((Clubs,Eight),(Clubs,King),(Gras,Ten),
         (Clubs,Nine));;
- : color * value = (Clubs,King)
```


Datatypes can be recursive:

```
type sequence = End | Next of (int * sequence)

# Next (1, Next (2, End));;
- : sequence = Next (1, Next (2, End))
```

Note the similarity to lists!