Script generated by TTT

Title: Seidl: Functional Programming and

Verification (07.12.2018)

Date: Fri Dec 07 08:44:42 CET 2018

Duration: 76:45 min

Pages: 17

→ 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'
```

203

Some of the Simplest Polymorphic Functions

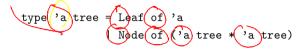
```
let compose f g x = f (g x)
let twice f x = f (f x)
let iter f g x = if g then x else ter f g (f x);

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'
```



3.5 Polymorphic Datatypes

User-defined datatypes may be polymorphic as well:



- ightarrow tree is called type constructor, because it allows to create a new type from another type, namely its parameter 'a.
- ightarrow In the right-hand side, only those type variables mya occur, which have been listed on the left.
- → The application of constructors to data may instantiate type variables:

204

Discussion

- The operator @ concatenates two lists.
- The implementation is very simple.
- Extraction is cheap.
- Insertion, however, requires as many calls of @ as the queue has elements.
- Can that be improved upon ??

206

Second Idea (cont.)

Insertion is in the second list:

Extracted are elements always from the first list:

Only if that is empty, the second list is consulted ...

Discussion

- Now, insertion is cheap!
- Extraction, however, can be as expensive as the number of elements in the second list ...
- Averaged over the number of insertions, however, the extra costs are only constant !!!

⇒ amortized cost analysis

216

3.7 Anonymous Functions

As we have seen, functions are data. Data, e.g., [1;2;3] can be used without naming them. This is also possible for functions:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- fun initiates an abstraction.
 This notion originates in the λ-calculus.
- -> has the effect of = in function definitions.
- Recursive functions cannot be defined in this way, as the recurrent occurrences in their bodies require names for reference.

 $\lambda f \times y \rightarrow f \times (fy)$

Discussion

- Now, insertion is cheap!
- Extraction, however, can be as expensive as the number of elements in the second list ...
- Averaged over the number of insertions, however, the extra costs are only constant !!!

⇒ amortized cost analysis

216

- Pattern matching can be used by applying match ... with for the corresponding argument.
- In case of a single argument, function can be considered ...



Alonzo Church, 1903-1995

218

- Pattern matching can be used by applying match ... with for the corresponding argument.
- In case of a single argument, <u>function</u> can be considered ...

3.7 Anonymous Functions

As we have seen, functions are data. Data, e.g., [1;2;3] can be used without naming them. This is also possible for functions:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- fun initiates an abstraction.
 This notion originates in the λ-calculus.
- -> has the effect of = in function definitions.
- Recursive functions cannot be defined in this way, as the recurrent occurrences in their bodies require names for reference.

217

Anonymous functions are convenient if they are used just once in a program. Often, they occur as arguments to functionals:

```
# map (fun x -> x*x) [1;2;3];;
- : int list = [1; 4; 9]
```

Often, they are also used for returning functions as result:

5.1 Exceptions

In case of a runtime error, e.g., division by zero, the Ocaml system generates an exception:

```
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".

Here, the exceptions    Division_by_zero,    Failure "tl"    and Invalid_argument "Char.chr"    are generated.
```

267

Another reason for an exception is an incomplete match:

```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
In this case, the exception Match_failure ("", 2, -9) is generated.
```

268