Why Functional Programming Matters

In the following a position statement by *John Hughes*, published in:

- Computer Journal 32(2), 98-107, 1989
- Research Topics in Functional Programming. D. Turner (Hrsg.), Addison Wesley, 1990
- http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html

"...an attempt to demonstrate to the "real world" that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are."

Typical Reasoning 1(4)

...functional programming owes its name to the facts that

- programs are composed of only functions
  - the "main program" is itself a function
  - it accepts its inputs as arguments and delivers its output as result
  - it is defined in terms of other functions, which themselves are defined by other functions (eventually by primitive functions)

Typical Reasoning 2(4)

Benefits and characteristics of functional programming. A common summary:

Functional programs are...

- free of assignments and side-effects
- function calls have no effect except of computing their result
- functional programs are thus free of a major source of bugs
- the evaluation order of expressions is irrelevant, expressions can be evaluated any time
- programmers are free from specifying the control flow explicitly
- expressions can be replaced by their value and vice versa, programs are *referentially transparent*
- functional programs are thus easier to cope with mathematically (e.g. for proving their correctness)

Typical Reasoning 3(4)

...the "default"-list of benefits and characteristica of functional programming yields

- essentially an "is-not"-characterization
  - *"It says a lot about what functional programming is* **not** (*it has no assignments, no side effects, no flow of control*) *but not much about what it is."*

Typical Reasoning 4(4)

No hard facts providing evidence for "real" benefits?

Yes, there are. Often heard e.g.:

- Functional programs are
  - a magnitude of order smaller than conventional programs
  - functional programmers are thus much more productive

But why? Justifyable by the benefits from the default catalogue? By dropping features? Hardly. Not convincing.

Conclusion

- The default catalogue is not satisfying
- We need a positive characterization of the principal nature of
  - functional programming and its strengths and
  - what makes up a "good" functional program

Towards a Positive Characterization... 1(2)

Analogue: Structural vs. non-structural programming

Structural programs are

- free of goto-statements ("goto considered harmful")
- blocks are free of multiple entries and exits
- easier to cope with mathematically than unstructured programs

Essentially an "is-not"-characterization, too...

Towards... 2(2)

Conceptually more important...

Structural programs are

- are designed modularly in distinction to non-structured programs
- Structural programming is more efficient/productive for this reason
  - Small modules are easier and faster to write and to maintain
  - Re-use becomes simpler
  - Modules can be tested independently

Note: Dropping goto-statements is not an essential source of productivity gain.

- Absence of gotos supports "*programming in the small*"
- Modularity supports "*programming in the large*"

Thesis

- The expressive power of a language, which supports modular design, depends much on the power of the concepts and primitives allowing to combine solutions of subproblems to the solution of the overall problem. (Keyword: *glue*). (Example: making of a chair)
- Functional programming provides two new, especially powerful means (“*glues*”) for this purpose:
  1. *Higher order functions* (*functionals*)
  2. *lazy evaluation*

  Modularization and re-use offer thus even *conceptually* (and not just technically (lexical scoping, separate compilation, etc.)) new opportunities and become much easier to apply
- Modularization (smaller, simpler, more general) is the guideline, which should be used by functional programmers for guidance

I Glueing Functions Together...

Syntax in the flavour of Miranda (TM):

- Lists

```
listof X ::= nil | cons X (listof X)
```

- Abbreviations

```
[]          short for     nil
[1]         short for     cons 1 nil
[1,2,3]     short for     cons 1 (cons 2 (cons 3 nil)))
```

- Adding the elements of a list

```
sum nil             = 0
sum (cons num list) = num + sum list
```


Observation

```
                        +---+
sum nil               = | 0 |
                        +---+
                               +---+
sum (cons num list) =  num     | + |  sum list
                               +---+
```

...the computation of a sum can be decomposed into modules by properly combining a general pattern of recursion and a set of more specific operations (see frames above).

```
sum = reduce add 0
where
  add x y = x+y
```

...revealing the definition of `reduce` almost immediately:

```
(reduce f x) nil        = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```


Immediate Benefits

Without any further programming effort we obtain...

- Computing the product of the elements of a list

```
product = reduce multiply 1
                 where multiply x y = x*y
```

- Test, if an element of a list equals “true”

```
anytrue = reduce or false
```

- Test, if all elements of a list equal “true”

```
alltrue = reduce and true
```


Intuition

The call `reduce f a` can be understood such that in a list of elements all occurrences of

- `cons` are replaced by `f` and of
- `nil` by `a`

Example:

```
reduce add 0:
  cons 1 (cons 2 (cons 3 nil))
          -->  add 1 (add 2 (add 3 0)) = 6

reduce multiply 1:
  cons 1 (cons 2 (cons 3 nil))
          -->  multiply 1 (multiply 2 (multiply 3 1)) = 6
```


More Applications 1(4)

- Observation
  `reduce cons nil` copies a list of elements
- This allows: `append a b = reduce cons b a`

Example:

```
append [1,2] [3,4] = reduce cons [3,4] [1,2]
                   = (reduce cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4])
                       -- replacement of cons by cons and
                       -- of nil by [3,4]
                   = [1,2,3,4]
```


More Applications 2(4)

- Copying each element of a list

```
doubleall = reduce doubleandcons nil
     where doubleandcons num list = cons (2*num) list
```

- Further step of modularization

```
doubleandcons = fandcons double
     where double n = 2*n
           fandcons f el list = cons (f el) list
```


More Applications 3(4)

- After another step of modularization

  `fandcons f = cons . f`

  where “.” denotes the composition of functions:
  `(f . g) h = f (g h)`

Illustration:

```
fandcons f el = (cons . f) el
              = cons (f el)
```

This yields as desired:

```
fandcons f el list = cons (f el) list
```

More Applications 4(4)

- Eventually, we thus obtain:
  `doubleall = reduce (cons . double) nil`
- Another step of modularization leads us to `map`

```
doubleall = map double
     where map f = reduce (cons . f ) nil
```

After this preparing steps it is just as well possible:

- To add the elements of a matrix:
  `summatrix = sum . map sum`

Intermediate Conclusion 1

By decomposion (modularization) of a simple function (`sum` in the example) as combination of

- a higher order function and
- some simple specific functions as arguments

we obtained a program frame (`reduce`), which allows us to implement many functions on lists without any further programming effort.

Generalizations to more complex data structures 1(2)

Trees

`treeof X ::= node X (listof (treeof X))`

Example:

```
node 1                                              1
   (cons (node 2 nil)                              / \
         (cons (node 3                            2   3
                    (cons (node 4 nil) nil))          |
               nil))                                  4
```


Generalizations... 2(2)

Analogously to `reduce` on lists we introduce a functional `redtree` on trees:

```
redtree f g a (node label subtrees) =
        f label (redtree' f g a subtrees)
where
   redtree' f g a (cons subtree rest) =
              g (redtree f g a subtree) (redtree' f g a rest)
   redtree' f g a nil = a
```


Applications 1(3)

- To add the labels of the leaves of a tree
  `sumtree = redtree add add 0`

Illustrated by means of an example:

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
= 10
```


Applications 2(3)

- Generating a list of all labels occurring in a tree
  `labels = redtree cons append nil`

Illustrated by means of an example:

```
cons 1
     (append (cons 2 nil)
             (append (cons 3
                           (append (cons 4 nil) nil))
                      nil))
= [1,2,3,4]
```


Applications 3(3)

- A function `maptree` on trees complementing the function `map` on lists
  `maptree f = redtree (node . f) cons nil`

Intermediate Conclusion 2 1(2)

- The expressiveness of the preceding examples is a consequence of combining
  - a higher order function and
  - a specific specializing function
- Once the higher order function is implemented, lots of further functions can be implemented almost without any effort

Intermediate Conclusion 2 2(2)

- *Lesson learnt*: Whenever a new data type is introduced, implement first a higher order function allowing to process (e.g., visiting each component of a structured data value such as nodes in a graph or tree) values of this type.
- *Benefits*: Manipulating elements of this data type becomes easy and knowledge about this data type is "localized".
- *Look&feel*: Whenever new data structures demand new control structures, then these control structures can easily be added following the methodology used above (to some extent this resembles the concepts known from conventional extensible languages)

II Glueing Programs Together

If `f` and `g` are programs, then also

```
g . f
```

is a program. Applied to the input `input`, it yields the output

```
g (f input)
```

- Possible convential implementation (glue): communication via files
- Possible problems
  - Temporary files are often too large
  - `f` might not terminate

More about lazy evaluation as a glue...

...next lecture!

This will be held (because of the Easter Holiday from April 2 - 14, 2007) on...

- Thu, April 19, 2007, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8

First assignment...

- Please check out the homepage of the course for details.