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: Structured vs. non-structured programming

Structured 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...

Structured programs are...

- designed modularly in distinction to non-structured programs
- Structured programming is more efficient/productive for this reason
  - Small modules are easier and faster to write and to maintain
  - Re-use becomes easier
  - 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

In the Following

- I Glueing Functions Together
  ⇝ The clou: *Higher-order functions*
- II Glueing Programs Together
  ⇝ The clou: *Lazy evaluation*

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 *some* 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

Functional Glue

Lazy evaluation offers a more elegant remedy.

As a glue, it allows:

- Decomposition of a problem into a
  - *generator* and a
  - *selector*

  component.

Intuition:

- The generator component "runs as little as possible" until it is terminated by the selector component.

Example 1: Computing Square Roots

Computing Square Roots (according to Newton-Raphson)

Given: N   Sought: squareRoot(N)

Iteration formula:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

Justification: If converging to some limit `a`, we have:

```
   a = (a + N/a) / 2
=> 2a = a + N/a
   a = N/a
   a*a = N
   a = squareRoot(N)
```


Compare this...

...with a typical imperative (Fortran-) program:

```
C     N is called ZN here so that it has the right type
         X = A0
         Y = A0 + 2.*EPS
C     The value of Y does not matter so long as ABS(X-Y).GT.EPS
100      IF (ABS(X-Y).LE.EPS) GOTO 200
         Y = X
         X = (X + ZN/X) / 2.
         GOTO 100
200      CONTINUE
C     The square root of ZN is now in X
```

$\rightsquigarrow$ essentially monolithic, not divisible.

The Functional Version 1(4)

Computing the next approximation

```
next N x = (x + N/x) / 2
```

Denoting this function `f`, we are interested in computing the sequence of approximations:

```
[a0, f a0, f(f a0), f(f(f a0)), ...]
```


The Functional Version 2(4)

The function `repeat` computes this (possibly infinite) sequence of approximations. It is the *generator* component in this example:

```
repeat f a = cons a (repeat f (f a))
```

Applying `repeat` to the arguments `next N` and `a0` yields the desired sequence of approximations:

```
repeat (next N) a0
```

The Functional Version 3(4)

Note: The evaluation of

```
repeat (next N) a0
```

does not terminate!

Remedy: ...computing `squareroot N` up to a given tolerance `eps` > 0. Instrumental is: the *selector* component.

Implementation:

```
within eps (cons a (cons b rest))
    = b,                          if abs(a-b) <= eps
    = within eps (cons b rest), otherwise
```

Still to do: Combining the components/modules:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

$\rightsquigarrow$ We are done.

The Functional Version 4(4)

Summing up:

- `repeat`... generator component:
  `[a0, f a0, f(f a0), f(f(f a0)), ...]`
  ...potentially infinite, no limit on the length
- `within`... selector component:
  f^i `a0` with `abs(`f^i `a0 -` f^{i+1} `a0) <= eps`
  ...lazy evaluation ensures that the selector function is applied eventually $\Rightarrow$ termination!

Note: Intuitively, lazy evaluation ensures that both programs (generator and selector) run in strict synchronization.

Towards the Re-Use of Modules

Next, we want to want to ;rovide evidence that

- generator
- selector

can indeed be considered modules, which can easily be re-used.

We are going to start with the re-use of the module *generator*...

Evidence of Modularity: Variants

Consider another criterion for termination:

- ...instead of awaiting the difference of successive approximations to approach zero (`<= eps`), await their ratio to approach one (`<= 1+eps`)

Implementation:

```
relative eps (cons a (cons b rest))
    = b,         if abs(a-b) <= eps * abs b
    = relative eps (cons b rest), otherwise
```

Still to do: (re-) composition of the components/modules:

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

$\rightsquigarrow$ We are done.

Note the Re-Use

...of the module *generator* in the previous example:

- The *generator*, i.e., the "module" computing the sequence of approximations has been re-used unchanged.

Next, we want to re-use the module *selector*...

Example 2: Numerical Integration

Numerical Integration

Given: A real valued function `f` of one real argument; two end-points `a` und `b` of an interval

Sought: The area under `f` between `a` and `b`

Naive Implementation:

...supposed that the function `f` is roughly linear between `a` und `b`.

```
easyintegrate f a b = (f a + f b) * (b-a) / 2
```

...sufficiently precise at most for very small intervals.

Refinements 1(4)

Idea

- Halve the interval, compute the areas for both subintervals according to the previous formula, and add the two results
- Continue the previous step repeatedly

The function `integrate` implements this strategy:

```
integrate f a b = cons (easyintegrate f a b)
                       map addpair (zip (integrate f a mid)
                                        (integrate f mid b)))
     where mid = (a+b)/2
```

Reminder:

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```


Refinements 2(4)

- `integrate` is sound but inefficient (redundant computations of `f a`, `f b`, and `f mid`

The following version of `integrate` is free of this deficiency

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa+fb)*(b-a)/2)
                         (map addpair (zip (integ f a m fa fm)
                                           (integ f m b fm fb)))
            where m = (a+b)/2
                  fm = f m
```

Refinements 3(4)

Apparently, the evaluation of

`integrate f a b`

does not terminate!

Remedy: ...computing `integrate f a b` up to some limit `eps` > 0.

Implementation:

```
Variant A:    within eps (integrate f a b)
Variant B:    relative eps (integrate f a b)
```


Refinements 4(4)

Summing up...

- Generator component:
  `integrate`
  ...potentially infinite, no limit on the length
- Selector component:
  `within`, `relative`
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ termination!

Note the Re-Use

...of the module *selector* in the previous example:

- The *selector*, i.e., the "module" picking the solution from the stream of approximate solutions has been re-used unchanged.

Again, *lazy evaluation* was the key to synchronize the generator and selector module!

Example 3: Numerical Differentiation

Numerical Differentiation

Given: A real valued function `f` of one real argument; a point `x`

Sought: The slope of `f` at point `x`

Naive Implementation:

...supposed that the function `f` between `x` and `x+h` does not "curve much"

```
easydiff f x h = (f (x+h) - f x) / h
```

...sufficiently precise at most for very small values of `h`.

Refinements 1(2)

Generate a sequence of approximations getting successively "better"

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

Selecting a sufficiently precise approximation

```
within esp (differentiate h0 f x)
```

⇝ Assignment

Conclusion 1(4)

The composition pattern, which in fact is common to all three examples becomes apparent again. It consists of

- generator (not limited itself!) and
- selector (ensuring termination thanks to lazy evaluation!)

Conclusion 2(4)

Thesis

- ...modularity is the key to *programming in the large*

Observation

- ...just modules (i.e., the capability of decomposing a problem) do not suffice
- ...the benefit of decomposing a problem into modular subproblems depends much on the capabilities for the *combination* of modules (glue!)
- ...the availability of proper glue is substantial!

Conclusion 3(4)

Fact

- Functional programming offers two new kinds of glue
  - *Higher-order functions*
  - *Lazy evaluation*
- Higher-order functions and lazy evaluation allow substantially new exciting modular decompositions of problems (by offering elegant composition means) as here given evidence by an array of impressive examples
- In essence, it it the superior glue, which makes functional programs to be written so concisely and elegantly

Conclusion 4(4)

Guideline

- Functional programmers should strive for adequate modularization and generalization
  - Especially, if a portion of a program looks ugly or appears to be too complex
- Functional programmers should expect that
  - *higher-order functions* and
  - *lazy evaluation*

  are the tools for doing this

Lazy vs. Eager Evaluation

The final conclusion of John Hughes...

- In view of the previous arguments...
  - The benefits of lazy evaluation as a glue is so evident that lazy evaluation is too important to make it a *second-class citizen*.
  - Lazy evaluation is possibly the most powerful glue functional programming has to offer.
  - Access to such a powerful means should not airily be dropped.

Worthwhile too...

...the examination of the following papers:

- Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. ACM Computing Surveys, Vol. 21, No. 3, 359-411, 1989.
- Phil Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.
- Simon Peyton Jones. *Wearing the Hair Shirt – A Retrospective on Haskell*. Invited Keynote Presentation at the 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.
  Slides: `http://research.microsoft.com/Users/simonpj/papers/haskell-retrospective/index.html`

Next lecture...

- Thu, March 13, 2008, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8