Why Functional Programming Matters

Starting softly with a position statement by *John Hughes*, based on an internal 1984 memo at Chalmers Univ., slightly revised 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."

Introductory Statement

As a matter of fact...

- Software is becoming more and more complex
- Hence: Structuring software well becomes paramount
- Well-structured software is more easily to write, to debug, and to be re-used

Claim

- Conventional languages place conceptual limits on the way problems can be modularized
- Functional languages push these limits back
- Fundamental: *Higher-order functions* and *lazy evaluation*

Next: Providing evidence for this claim...

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 the program's input as its arguments and delivers the program's output as its result
  - it is defined in terms of other functions, which themselves are defined in terms of still more functions (eventually by primitive functions)

Typical Reasoning 2(4)

Advantages 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 standard list of characteristics and advantages of functional programming yields

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

Typical Reasoning 4(4)

Aren't there any hard facts providing evidence for "real" advantages?

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? Is it justifiable by the advantages of the standard catalogue? By dropping features? Hardly. This is not convincing.

Reminds more to a medieval monk who denies himself the pleasures of life in the hope of getting virtuous...

Conclusion

- The standard catalogue is not satisfying
  - It does not provide help in exploiting the power of functional languages
    - Programs cannot be written which are particularly lacking in assignment statements, or particularly referentially transparent
  - It does not provide a yardstick of program quality, thus no model to strive for
- We need a positive characterization of the vital nature of
  - functional programming, of its strengths
  - what makes a "good" functional program, of what a functional programmer should strive for

Towards a Positive Characterization... 1(2)

Analogue: Structured vs. non-structured programming

Structured programs are

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

Essentially this is also a negative "is-not"-characterization...

Towards... 2(2)

Conceptually more important:

Structured programs are...

- designed modularly in contrast 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 expressiveness of a language that 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 *glues*:
  1. **Higher-order functions** (**functionals**)
  2. **Lazy evaluation**

  They offer *conceptually* new opportunities for modularization and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.
- Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming

In the Following

- I Glueing functions together
  $\rightsquigarrow$ The clou: *Higher-order functions*
- II Glueing programs together
  $\rightsquigarrow$ The clou: *Lazy evaluation*

I Glueing Functions Together

Syntax in the flavour of Miranda (TM):

- Lists

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

- Abbreviations for convenience

```
[]          means     nil
[1]         means     cons 1 nil
[1,2,3]     means     cons 1 (cons 2 (cons 3 nil)))
```

Motivating example: Adding the elements of a list

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

Observation

Only the framed parts are specific to computing a sum...

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

...i.e., computing a sum of values can be modularly decomposed by properly combining a general recursion pattern and a set of more specific operations (see framed parts above).

Realization

1. Adding the elements of a list

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

...which reveals the definition of `reduce` almost immediately:

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

Recall

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


Immediate Benefit – Re-use

Without any further programming effort we obtain implementations for...

2. Computing the product of the elements of a list

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

3. Test, if *some* element of a list equals "true"

```
anytrue = reduce or false
```

4. 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`
- `nil` by `a`

Examples:

```
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(5)

Observation: `reduce cons nil` copies a list of elements

This allows:

5. Concatenation of lists

```
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))
                   = { replacing cons by cons and nil by [3,4] }
                     cons 1 (cons 2 [3,4])
                   = cons 1 (cons 2 (cons 3 (cons 4 nil)))
                   = [1,2,3,4]
```


More Applications 2(5)

6. Doubling each element of a list

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


More Applications 3(5)

The function `doubleandcons` can be modularized further...

- First step

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

- Second step

```
fandcons f = cons . f
```

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

For checking correctness consider...

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

which yields as desired:

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

More Applications 4(5)

Eventually, we thus obtain:

6a. Doubling each element of a list

```
doubleall = reduce (cons . double) nil
```

Another step of modularization leads us to `map`

6b. Doubling each element of a list

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

where `map` applies any function `f` to all the elements of a list.

More Applications 5(5)

After these preparative steps it is possible just as well:

7. Adding the elements of a matrix

```
summatrix = sum . map sum
```

Homework: Think about how summatrix works...

1st Intermediate Conclusion

By decomposing (modularizing) and representing a simple function (`sum` in the example) as a 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
```

Note, `redtree` takes 3 arguments (`f`, `g`, `a`)

- The first one to replace `node` with
- The second one to replace `cons` with
- The third one to replace `nil` with

Applications 1(3)

1. Adding the labels of the leaves of a tree

```
sumtree = redtree add add 0
```

Using the tree introduced previously, we obtain:

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


Applications 2(3)

2. 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)

3. A function `maptree` on trees replicating the function `map` on lists

```
maptree f = redtree (node . f) cons nil
```


2nd Intermediate Conclusion 1(2)

- The elegance 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 further effort!

2nd Intermediate Conclusion 2(2)

- *Lesson learnt*: Whenever a new data type is introduced, implement first a higher-order function allowing to process values of this type (e.g., visiting each component of a structured data value such as nodes in a graph or tree).
- *Benefits*: Manipulating elements of this data type becomes easy and knowledge about this data type is locally concentrated.
- *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).

Reminder

Thesis

- The expressiveness of a language that 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 *glues*:
  1. **Higher-order functions** (**functionals**)
  2. **Lazy evaluation**

  They offer *conceptually* new opportunities for modularization and reuse (beyond the more technical ones of lexical scoping, separate compilation, etc.), and make them more easily to achieve.
- Modularization (smaller, simpler, more general) is the guideline, which should be followed by functional programmers in the course of programming

Reminder (Cont'd)

We did talk about...

- Higher-order functions as glue for *glueing functions together*

We still have to talk about...

- Lazy evaluation as glue for *glueing programs together*

II Glueing Programs Together

Recall: A complete functional program is a function from its input to its output.

If `f` and `g` are (such) programs, then also

```
g . f
```

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

```
g (f input)
```

- Possible implementation using conventional glue: communication via files
  - Possible problems
    - Temporary files are often too large
    - `f` might not terminate

Functional Glue

Lazy evaluation allows a more elegant approach:

- Decomposing a problem into a
  - *generator* and a
  - *selector*

  component, which are then glued together.

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 the approximations converge to some limit `a`, we have:

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

I.e., `a` stores the value of the square root of `N`.

Compare this...

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

```
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
```

⇝ essentially monolithic, not decomposable.

The Functional Version 1(4)

Computing the next approximation from the previous one:

```
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:

Generator:

```
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. Crucial: The *selector* component implemented by:

Selector:

```
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)
```

$\leadsto$ 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:
  $\texttt{f}^i$ `a0` with `abs(`$\texttt{f}^i$ `a0 -` $\texttt{f}^{i+1}$ `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 strictly synchronized.

Towards the Re-Use of Modules

Next, we want to provide evidence that

- generator
- selector

can indeed be considered modules that 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`)

New **Selector**:

```
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-)combining of the components/modules:

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

$\leadsto$ 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 endpoints `a` und `b` of an interval

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

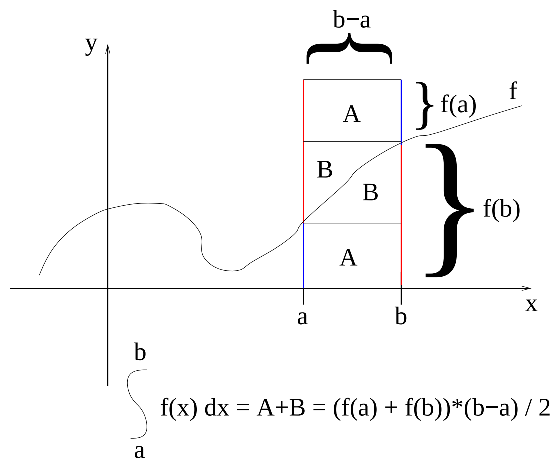
Naive Implementation:

...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.

Illustration



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:

Generator:

```
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.

Two **Selectors**:

```
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 $\Rightarrow$ 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* is 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

Generate a sequence of approximations getting successively "better":

Generator:

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

Select a sufficiently precise approximation:

Selector:

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

Implementing the selector: Homework

Conclusion 1(4)

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

- generator (usually looping!) 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 modularly decomposing a problem into subproblems depends much on the capabilities for *glueing* the modules together
- ...the availability of proper *glue* is essential!

Conclusion 3(4)

Fact

- Functional programming offers two new kinds of glue
  - *Higher-order functions* (glueing functions)
  - *Lazy evaluation* (glueing programs)
- 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 simple, yet impressive examples
- In essence, it is the superior glue, which makes functional programs to be written so concisely and elegantly (not the absence of assignments, etc.)

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 achieving 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 are 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...

...studying 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 course meeting...

- Thu, 17 March 2011, 4.15 p.m. to 5.45 p.m., lecture room on the groundfloor of the Institutsgebäude, Argentinierstr. 8