Reminder

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

Reminder (Cont'd)

We did talk about...

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

We did not yet talk about...

- Lazy evaluation as glue for *glueing programs together*

I Glueing Functions Together

See part I of this lecture.

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)
```

A possible conventional implementation (glue): communication via files

Possible problems of such an implementation:

- 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
```


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)
```


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 ⇒ termination!

Evidence of Modularity: Variants

Consider another stop criterion:

- ...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)
```

Note: The generator, i.e., the "module" computing the sequence of approximations can be reused unchanged.

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)

Note: 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 ⇒ Terminierung!

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)
```


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 do not suffice
- ...the benefit of decomposing a problem into modular sub-problems 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

Reconsidering...

- In view of the previous arguments...
  - The benefits of lazy evaluation as 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 frivolously 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`

Last but not least...

Next lecture...

- Thu, April 26, 2007, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8

Second assignment...

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