

---

## Infinite Lists: Programming with Streams

The following presentation is based on...

- Chapter 14  
Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Chapter 17  
Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.

---

## Streams

*Jargon*

*Stream* ...synonymous to *infinite list*  
synonymous to *lazy list*

Streams...

- (in combination with lazy evaluation) allow to solve many problems elegantly, concisely, and efficiently
- are a source of hassle if applied inappropriately

More on this on the following slides...

---

## Streams

*Convention*

Instead of introducing a polymorphic data type `Stream`...

```
data Stream a = a :* Stream a
```

...we will model streams by ordinary lists waiving the usage of the empty list `[ ]`.

This is motivated by:

- Convenience/Adequacy ...many pre-defined (polymorphic) functions on lists can be reused this way, which otherwise would have to be defined on the new data type `Stream`

---

## Some Examples of Streams

- *Built-in Streams in Haskell*

```
[3 ..] = [3,4,5,6,7,...  
[3,5 ..] = [3,5,7,9,11,...
```

- *User-defined recursive lists (Streams)*

The infinite lists of "twos"

```
2,2,2,...
```

In Haskell this can be realized...

- using list comprehension: `[2..]`
- as a recursive stream: `twos = 2 : twos`

*Illustration*

```
twos => 2 : twos  
=> 2 : 2 : twos  
=> 2 : 2 : 2 : twos  
=> ...
```

...`twos` represents an infinite list; or more concisely, a *stream*

---

## Functions on Streams

```
head :: [a] -> a
head (x:_) = x
```

### Application

```
head twos
=> head (2 : twos)
=> 2
```

*Note:* Normal-order reduction (resp. its efficient implementation on variant *lazy evaluation*) ensures termination (in this example). I.e., the infinite sequence of reductions...

```
head twos
=> head (2 : twos)
=> head (2 : 2 : twos)
=> head (2 : 2 : 2 : twos)
=> ...
```

...is thus excluded.

---

## Reminder

...whenever there is a terminating reduction sequence of an expression, then normal-order reduction terminates (Church/Rosser-Theorem)

- Normal-order reduction corresponds to *leftmost-outermost* evaluation

*Note:* Considering the function...

```
ignore :: a -> b -> b
ignore a b = b
```

in both expressions

- ignore twos 42
- twos 'ignore' 42

the leftmost-outermost operator is given by the call ignore.

---

## Functions on Streams: More Examples

```
addFirstTwo :: [Integer] -> Integer
addFirstTwo (x:y:zs) = x+y
```

### Application

```
addFirstTwo twos => addFirstTwo (2:twos)
=> addFirstTwo (2:2:twos)
=> 2+2
=> 4
```

---

## Further Examples on Streams

- User-defined recursive lists/streams

```
from :: Int -> [Int]
from n = n : from (n+1)
```

```
fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```

### Application

```
from 42 => [42, 43, 44, ...]
```

```
fromStep 3 2 => 3 : fromStep 5 2
=> 3 : 5 : fromStep 7 2
=> 3 : 5 : 7 : fromStep 9 2
=> ...
```

---

## Further Examples

- The powers of an integer...

```
powers :: Int -> [Int]
powers n = [n^x | x <- [0 ..]]
```

- More general: The prelude function iterate...

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

The function iterate yields the stream

```
[x, f x, (f . f) x, (f . f . f) x, ..
```

---

## Prime Numbers: The Sieve of Eratosthenes 1(4)

### Intuition

1. Write down the natural numbers starting at 2.
2. The smallest number not yet cancelled is a prime number. Cancel all multiples of this number
3. Repeat Step 2 with the smallest number not yet cancelled.

### Illustration

```
Step 1:      2 3 4 5 6 7 8 9 10 11 12 13...
Step 2:      2 3   5   7   9   11   13...
("with 2")
Step 2:      2 3   5   7           11   13...
("with 3")
...
```

---

## Prime Numbers: The Sieve of Eratosthenes 2(4)

The sequence of prime numbers...

```
primes :: [Int]
primes = sieve [2 ..]
```

```
sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0 ]
```

---

## Prime Numbers: The Sieve of Eratosthenes 3(4)

### Illustration ...by manual evaluation

```
primes
=> sieve [2 ..]
=> 2 : sieve [ y | y <- [3 ..], mod y 2 > 0 ]
=> 2 : sieve (3 : [ y | y <- [4 ..], mod y 2 > 0 ]
=> 2 : 3 : sieve [ z | z <- [ y | y <- [4 ..], mod y 2 > 0 ],
                mod z 3 > 0 ]
=> ...
=> 2 : 3 : sieve [ z | z <- [5, 7, 9 ..], mod z 3 > 0 ]
=> ...
=> 2 : 3 : sieve [5, 7, 11,...]
=> ...
```

---

## Prime Numbers: The Sieve of Eratosthenes 4(4)

- *Application*

```
member primes 7    ...yields "True"

but

member primes 6    ...does not terminate!

where
```

```
member :: [a] -> a -> Bool
member []    y = False
member (x:xs) y = (x==y) || member xs y
```

- *Question(s)*: Why? How can primes be embedded into a context allowing us to detect if a specific argument is prime or not? (Homework)

---

## Random Numbers 1(2)

Generating a sequence of (pseudo-) random numbers...

```
nextRandNum :: Int -> Int
nextRandNum n = (multiplier*n + increment) 'mod' modulus
```

```
randomSequence :: Int -> [Int]
randomSequence = iterate nextRandNum
```

Choosing

```
seed          = 17489          increment = 13849
multiplier    = 25173          modulus    = 65536
```

we obtain the following sequence of (pseudo-) random numbers

[17489, 59134, 9327, 52468, 43805, 8378, ...

ranging from 0 to 65536, where all numbers of this interval occur with the same frequency.

---

## Random Numbers 2(2)

Often one needs to have random numbers within a range  $p$  to  $q$  inclusive,  $p < q$ .

This can be achieved by scaling the sequence.

```
scale :: Float -> Float -> [Int] -> [Float]
scale p q randSeq = map (f p q) randSeq
  where f :: Float -> Float -> Int -> Float
        f p q n = p + ((n * (q-p)) / (modulus-1))
```

*Application*

```
scale 42.0 51.0 randomSequence
```

---

## Principles of Modularization

...related to streams

- The *Generator/Selector* Principle  
...e.g. Computing the square root, the Fibonacci numbers
- The *Generator/Transformer* Principle  
...e.g. "scaling" random numbers

---

## More on Recursive Streams

*Reminder* ...the sequence of Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...

is defined by

$fib : \mathbb{N} \rightarrow \mathbb{N}$

$$fib(n) =_{df} \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

---

## The Fibonacci Numbers 1(4)

We learned already...

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

...that a naive implementation as above is inacceptably inefficient.

---

## The Fibonacci Numbers 2(4)

*Illustration* ...by manual evaluation

fib 0 => 1 -- 1 call of fib

fib 1 => 1 -- 1 call of fib

fib 2 => fib 1 + fib 0  
=> 1 + 1  
=> 2 -- 3 calls of fib

fib 3 => fib 2 + fib 1  
=> (fib 1 + fib 0) + 1  
=> (1 + 1) + 1  
=> 3 -- 5 calls of fib

---

## The Fibonacci Numbers 3(4)

```
fib 4 => fib 3 + fib 2
      => (fib 2 + fib 1) + (fib 1 + fib 0)
      => ((fib 1 + fib 0) + 1) + (1 + 1)
      => ((1 + 1) + 1) + (1 + 1)
      => 5 -- 9 calls of fib
```

```
fib 5 => fib 4 + fib 3
      => (fib 3 + fib 2) + (fib 2 + fib 1)
      => ((fib 2 + fib 1) + (fib 1 + fib 0))
          + ((fib 1 + fib 0) + 1)
      => (((fib 1 + fib 0) + 1) + (1 + 1)) + ((1 + 1) + 1)
      => (((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)
      => 8 -- 15 calls of fib
```

## The Fibonacci Numbers 4(4)

```

fib 8 => fib 7 + fib 6
=> (fib 6 + fib 5) + (fib 5 + fib 4)
=> ((fib 5 + fib 4) + (fib 4 + fib 3))
    + ((fib 4 + fib 3) + (fib 3 + fib 2))
=> (((fib 4 + fib 3) + (fib 3 + fib 2))
    + (fib 3 + fib 2) + (fib 2 + fib 1)))
    + (((fib 3 + fib 2) + (fib 2 + fib 1))
    + ((fib 2 + fib 1) + (fib 1 + fib 0)))
=> ... -- 60 calls of fib

```

...tree-like recursion (exponential growth!)

## Reminder: Complexity 1(3)

See P. Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, 2nd Edition (In German), 2003, Chapter 11.

Reminder ... $\mathcal{O}$  Notation

- Let  $f$  be a function  $f : \alpha \rightarrow \mathbb{R}^+$  with some data type  $\alpha$  as domain and the set of positive real numbers as range. Then the class  $\mathcal{O}(f)$  denotes the set of all functions which "grow slower" than  $f$ :

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ for some positive constant } c \text{ and all } n \geq N_0\}$$

## Reminder: Complexity 2(3)

Examples of common cost functions...

Code	Costs	Intuition: <i>input a thousandfold as large means...</i>
$\mathcal{O}(c)$	constant	... equal effort
$\mathcal{O}(\log n)$	logarithmic	...only tenfold effort
$\mathcal{O}(n)$	linear	...also a thousandfold effort
$\mathcal{O}(n \log n)$	" $n \log n$ "	...tenthousandfold effort
$\mathcal{O}(n^2)$	quadratic	...millionfold effort
$\mathcal{O}(n^3)$	cubic	...billiardfold effort
$\mathcal{O}(n^c)$	polynomial	... gigantic much effort (for big $c$ )
$\mathcal{O}(2^n)$	exponential	...hopeless

## Reminder: Complexity 3(3)

...and the impact of growing inputs in practice in hard numbers:

n	linear	quadratic	cubic	exponential
1	1 $\mu$ s	1 $\mu$ s	1 $\mu$ s	2 $\mu$ s
10	10 $\mu$ s	100 $\mu$ s	1 ms	1 ms
20	20 $\mu$ s	400 $\mu$ s	8 ms	1 s
30	30 $\mu$ s	900 $\mu$ s	27 ms	18 min
40	40 $\mu$ s	2 ms	64 ms	13 days
50	50 $\mu$ s	3 ms	125 ms	36 years
60	60 $\mu$ s	4 ms	216 ms	36 560 years
100	100 $\mu$ s	10 ms	1 sec	$4 * 10^{16}$ years
1000	1 ms	1 sec	17 min	very, very long...

---

## Remedy: Recursive Streams 1(4)

*Idea*

```
1 1 2 3 5 8 13 21...   Sequence of Fibonacci Numbers
1 2 3 5 8 13 21 34...   Remainder of the sequ. of F. Numbers
-----
2 3 5 8 13 21 34 55...   Remain. of the rem. of the seq. of F
```

Efficient implementation as a recursive stream

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

where

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

...reminds to Münchhausen's famous trick of "sich am eigenen Schopfe aus dem Sumpfe ziehen"

---

## Remedy: Recursive Streams 2(4)

```
fibs => 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 : ...
```

```
take 10 fibs => [1,1,2,3,5,8,13,21,34,55]
```

where

```
take :: Integer -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"
```

---

## Remedy: Recursive Streams 3(4)

Summing up

```
fib :: Integer -> Integer
fib n = last take n fibs
```

or even yet shorter

```
fib n = fibs!!n
```

Note:

- Also in this example...  
Application of the *Generator/Selector* Principle

---

## Remedy: Recursive Streams 4(4)

*Illustration* ...by manual evaluation (with add instead of zipWith (+) )

```
fibs => Replace the call of fibs by the body of fibs
      1 : 1 : add fibs (tail fibs)
=> // Replace both calls of fibs by the body of fibs
   1 : 1 : add (1 : 1 : add fibs (tail fibs))
                               (tail (1 : 1 : add fibs (tail fibs)))
=> // Application of tail
   1 : 1 : add (1 : 1 : add fibs (tail fibs))
                               (1 : add fibs (tail fibs))
=> ...
```

- *Observation*  
...the computational effort remains exponential this (naive) way!
- *Clou*  
...lazy evaluation: ...common subexpressions will not be computed multiple times!

## Illustration 1(3)

```
fibs => 1 : 1 : add fibs (tail fibs)

=> // Introducing abbreviations allows sharing of results
1 : tf // (tf reminds to "tail of fibs")
where tf = 1 : add fibs (tail fibs)
=> 1 : tf
where tf = 1 : add fibs tf

=> // Introducing abbreviations allows sharing
1 : tf
where tf = 1 : tf2 // (tf2 reminds to "tail of tail
// of fibs")
where tf2 = add fibs tf

=> // Unfolding of add
1 : tf
where tf = 1 : tf2
where tf2 = 2 : add tf tf2
```

## Illustration 2(3)

```
=> // Repeating the above steps
1 : tf
where tf = 1 : tf2
where tf2 = 2 : tf3 // (tf3 reminds to "tail of
// tail of tail of fibs")
where tf3 = add tf tf2

=> 1 : tf
where tf = 1 : tf2
where tf2 = 2 : tf3
where tf3 = 3 : add tf2 tf3

=> // tf is only used at one place and can thus be
// eliminated
1 : 1 : tf2
where tf2 = 2 : tf3
where tf3 = 3 : add tf2 tf3
```

## Illustration 3(3)

```
=> // Finally, we obtain successsively longer prefixes
// the sequence of Fibonacci numbers
1 : 1 : tf2
where tf2 = 2 : tf3
where tf3 = 3 : tf4
where tf4 = add tf2 tf3

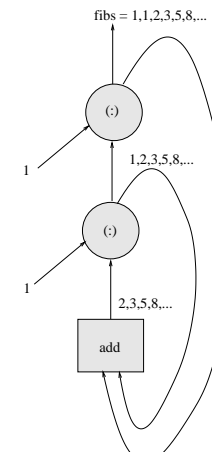
=> 1 : 1 : tf2
where tf2 = 2 : tf3
where tf3 = 3 : tf4
where tf4 = 5 : add tf3 tf4

// Note: eliminating where-clauses corresponds to
// garbage collection of unused memory by an implementation
=> 1 : 1 : 2 : tf3
where tf3 = 3 : tf4
where tf4 = 5 : add tf3 tf4
```

## Alternatively: Stream Diagrams

Problems on streams can often be considered and visualized as processes.

Considering the sequence of Fibonacci Numbers as an example...





## Another Example: A Client/Server Application

Interaction of a server and a client (e.g. Web server/Web browser)

```
client :: [Response] -> [Request]
server :: [Request] -> [Response]
```

```
reqs = client resps
resps = server reqs
```

### Implementation

```
type Request = Integer
type Response = Integer
```

```
client ys = 1 : ys // ...issues 1 as first request and then
                // each integer it receives from the server
server xs = map (+1) xs // ...adds 1 to each request it receives
```

## Client/Server Application (Cont'd. 1(2))

### Example

```
reqs => client resps
=> 1 : resps
=> 1 : server reqs

=> // Introducing abbreviations
  1 : tr
  where tr = server reqs
=> 1 : tr
  where tr = 2 : server tr
=> 1 : tr
  where tr = 2 : tr2
        where tr2 = server tr
```

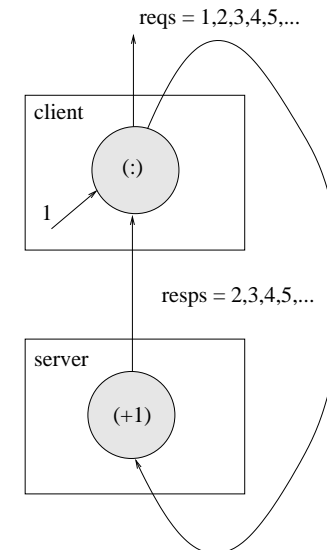
## Client/Server Application (Cont'd. 2(2))

```
=> 1 : tr
  where tr = 2 : tr2
        where tr2 = 3 : server tr2
=> 1 : 2 : tr2
  where tr2 = 3 : server tr2
=> ...
```

In particular

```
take 10 reqs => [1,2,3,4,5,6,7,8,9,10]
```

## The Client/Server Example as a Stream Diagram



---

## Overcoming Hassle... Lazy Patterns

Suppose, the client wants to check the first response...

```
client (y:ys) = if ok y then 1 : (y:ys)
                else error "Faulty Server"
```

```
where
ok y = True // Apparently a trivial predicate
```

The evaluation of...

```
reqs => client resps
      => client (server reqs)
      => client (server (client resps))
      => client (server (client (server reqs)))
      => ...
```

...does not terminate!

*The problem:*

Deadlock! Neither client nor server can be unfolded! Pattern matching is too "eager."

---

## Lazy Patterns 1(3)

Ad-hoc Remedy

```
client ys = 1 : if ok (head ys) then ys
              else error "Faulty Server"
```

- Replacing of pattern matching by an explicit usage of the selector function `head`
- Moving the conditional inside of the list

---

## Lazy Patterns 2(3)

Systematic remedy ...lazy patterns

- Syntax: ...preceding tilde (~)
- Effect: ...like using an explicit selector function; pattern-matching is deferred

```
client ~(y:ys) = 1 : if ok y then y:ys
                else error "Faulty Server"
```

Note ...even when using a lazy pattern the conditional must still be moved. But: selector functions are avoided!

---

## Lazy Patterns 3(3)

*Illustration* ...by manual evaluation

```
reqs => client resps
      => 1 : if ok y then y : ys
          else error "Faulty Server"
          where y:ys = resps
      => 1 : (y:ys)
          where y:ys = resps
      => 1 : resps
```

---

## Overcoming Hassle... Memo Tables

*Note* ...Dividing/Recognizing of common structures is limited

The below variant of the Fibonacci function...

```
fibsFn :: () -> [Integer]
fibsFn x = 1 : 1 : zipWith (+) (fibsFn ()) (tail (fibsFn ()))
```

...exposes again exponential run-time and storage behaviour!

Key word:

- *Space (Memory) Leak* ...the memory space is consumed so fast that the performance of the program is significantly impacted

---

## Illustration

```
fibsFn ()
=> 1 : 1 : add (fibsFn ()) (tail (fibsFn ()))
=> 1 : tf
   where tf = 1 : add (fibsFn ()) (tail (fibsFn ()))
```

The equality of `tf` and `tail(fibsFn())` remains undetected. Hence, the following simplification is not done

```
=> 1 : tf
   where tf = 1 : add (fibsFn ()) tf
```

In a special case like here, this is possible, but not in general!

---

## Memo Functions 1(4)

Memo functions (engl. *Memoization*)....

- The *concept* goes back to Donald Michie. “*“Memo” Functions and Machine Learning*”, *Nature*, 218, 19-22, 1968.
- *Idea*: Replace, where possible, the computation of a function according to its body by looking up its value in a table.

---

## Memo Functions 2(4)

- *Hence*: A memo function is an ordinary function, but stores for some or all arguments it has been applied to the corresponding results  $\rightsquigarrow$  Memo Tables.
- *Utility*: *Memo Tables* – allow to replace recomputation by table look-up  
Correctness: Referential transparency of functional programming languages

---

## Memo Functions 3(4)

Computing the Fibonacci Numbers using a memo function:

Preparation:

```
flist = [ f x | x <- [0 ..] ]
```

...where *f* is a function on integers. *Application*: Each call of *f* is replaced by a look-up in *flist*.

Considering the Fibonacci numbers as example:

```
flist = [ fib x | x <- [0 ..] ]
fib 0 = 1
fib 1 = 1
fib n = flist !! (n-1) + flist !! (n-2)
```

instead of...

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

---

## Memo Functions 4(4)

Conclusion...

- Memo Functions: Are meant to replace costly to compute functions by a table look-up
- Example ( $2^0$ ,  $2^1$ ,  $2^2$ ,  $2^3$ , ...):

```
power 0 = 1
power i = power (i-1) + power (i-1)
```

Looking-up the result of the second call instead of recomputing it requires only  $1+n$  calls of *power* instead of  $1+2^n$   
~> significant performance gain

---

## Memo Tables 1(2)

Memo functions/tables

```
memo :: (a -> b) -> (a -> b)
```

are used such that the following equality holds:

```
memo f x = f x
```

*Key word*: Referential transparency (in particular, absence of side effects!)

---

## Memo Tables 2(2)

The function `memo`...

- essentially the identity on functions but...
- `memo` keeps track on the arguments, it has been applied to and the corresponding results  
...motto: look-up a result which has been computed previously instead of recomputing it!
- Memo functions are not part of the Haskell standard, but there are nonstandard libraries
- Important design decision when implementing Memo functions: ...how many argument/result pairs shall be traced? (e.g. `memo1` for one argument/result pair)

*In the example*

```
mfibsFn :: () -> [Integer]
mfibsFn x = let mfibs = memo1 mfibsFn
            in 1 : 1 : zipWith (+) (mfibs ()) (tail (mfibs ()))
```

## More on Memo Functions...

...and their implementation

For example in...

- Chapter 19  
Anthony J. Field, Peter G. Harrison. *Functional Programming*, Addison-Wesley, 1988.

## Summary

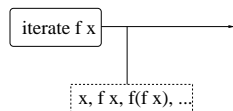
What are the reasons advocating the usage of streams (and lazy evaluation)?

- *Higher abstraction* ...limitations to finite lists are often more complex, while simultaneously unnatural
- *Modularization* ...together with lazy evaluation as evaluation strategy elegant possibilities for modularization become possible. Keywords are the *Generator/Selector* and the *Generator/Transformer* principle.

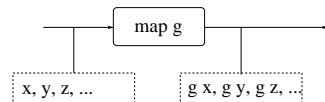
## Generator/Transformer Principle

Illustration...

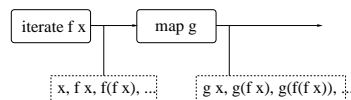
Generator



Transformer



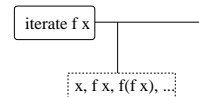
Combining Generator and Transformer



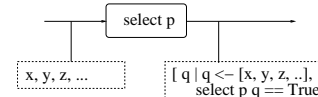
## Generator/Selector Principle

Illustration...

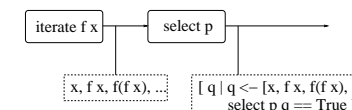
Generator



Selector/Filter



Combining Generator and Selector/Filter



---

## Invitation

The “Compilers and Languages” Group at the Institute of Computer Languages invites you to a talk by

**Prof. Dr. Robert Glück** on

*Is There a Fourth Futamura Projection?*

**Wednesday, 1 April 2009, 5:00 p.m.,**

Library E185.1, Argentinierstr. 8, 4th Floor (Centre).

Everybody is welcome!

For further details please check out

<http://www.complang.tuwien.ac.at/talks/Gluck2009-04-01>

---

## Next course meetings...

We will continue in April after the Easter holiday on...

- Thu, April 23, 2009, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8
- Fri, April 24, 2009, lecture time: 4.15 p.m. to 5.45 p.m., lecture hall EI 3a, 2nd floor, Gußhausstr. 25-29