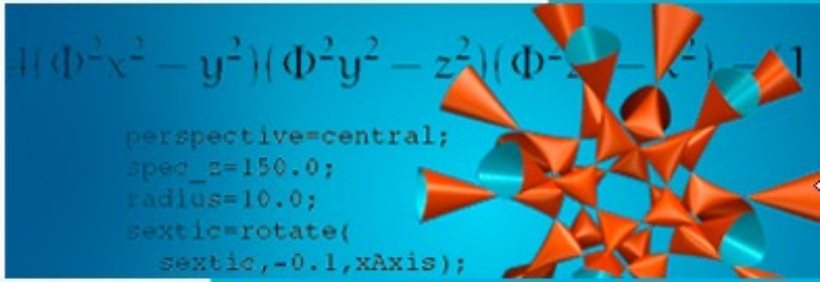
Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 14]

Generative recursion

- Design recipe for functions via algebraic data types
  - One auxiliary function per alternative
- Design recipe for functions using recursive data types
  - Structural recursion
- Design recipe cannot always be applied
  - Structure of the data does not match the breakdown of the problem

1st case study

- Based on move-ball function
- Calculate where the ball collides

- Simulation of the movement by recursive insertion of the function
  - (move-ball ball)
  - (move-ball (move-ball ball))
  - Etc.
- Recursion abort as soon as (collision current-ball) not "none"

1st case study

Does this function correspond to one of our design patterns?

```
; Ball -> Posn
; computes the position where the first
; collision of the ball occurs
(define (first-collision ball)
   (cond [(string=? (collision (ball-loc ball)) "none")
            (first-collision (move-ball ball))]
          [else (ball-loc ball)]))
```

Case differentiation and recursion are independent of the structure of the input type.

Argument for recursive call not generated by selector on the input value, but newly generated.

2nd case study

- Sorting a list of numbers
- List is a recursive data type
- According to the design recipe:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (cond
    [(empty? l) ...]
    [else ... (first l) ... (sort (rest l)) ...]))
```

2nd case study

- Sorting a list of numbers
- List is a recursive data type
- According to the design recipe:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (cond
    [(empty? l) ...]
    [else ... (first l) ... (sort (rest l)) ...]))
```

Basic case: The empty list is sorted.

Recursive case: Sorted insertion of (first l) in sorted list (sort (rest l))

2nd case study

- Sorting a list of numbers
- List is a recursive data type
- According to the design recipe:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (cond
    [(empty? l) empty]
    [else (insert (first l) (sort (rest l) ) )]))
```

Insertion Sort:

- Sorting algorithm with structural recursion.
- Not efficient

2nd case study

- Efficiency of Insertion Sort

- Given a function call
  (sort (list x-1 ... x-n))
- Then the expansion of this is
  (insert x-1 (insert x-2 ... (insert x-n empty) ...))

Max. n calculation steps

Max. n - 1 calculation steps

Max. 1 calculation step

- Efficiency of insert
  - Runs through the list until insertion position found
  - For backwards sorted input: run through completely each time

2nd case study

- Efficiency of Insertion Sort

- Calculation steps:
  $n + (n - 1) + \ldots 1 = n * (n + 1) / 2 = (n^2 + n) / 2$

- Efficient:
  - In the worst case, the number of calculation steps depends on the square of the length of the input list.

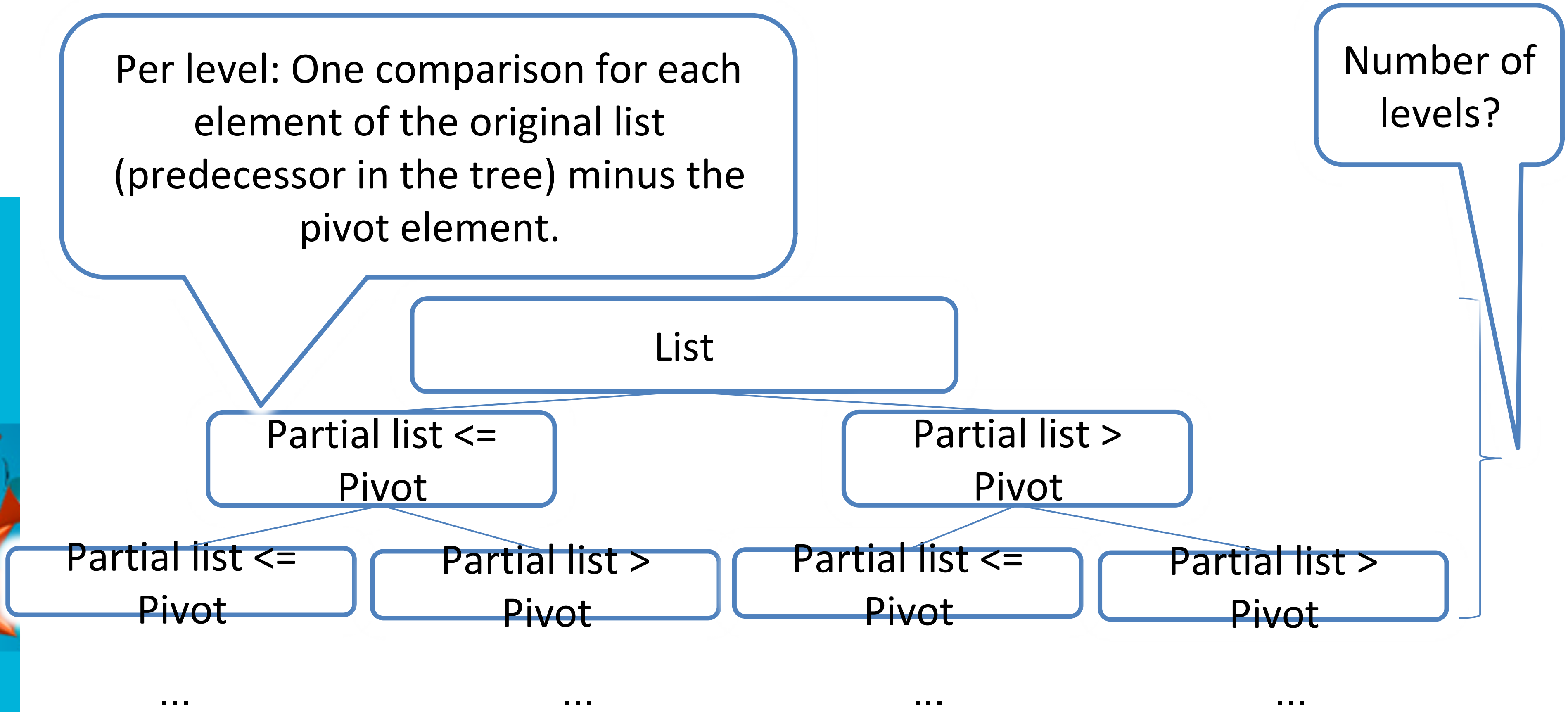
2nd case study

- Better efficiency possible
- In addition: better division into sub-problems
- Known algorithm: Quick Sort

2nd case study

- Quick Sort
  - Selection of any element of the list (called "pivot element")
  - Generate:
    - A list with elements <= pivot element
    - A list of elements > Pivot element
  - Recursive call of Quick Sort with both lists
  - Merge both sorted lists with pivot element in the middle

2nd case study



2nd case study

- Number of levels (recursion depth)
  - Depending on the choice of pivot elements
  - Worst case:
    A sublist contains exactly the pivot element: n
  - Best case:
    Division into two equally sized sublists: log(n)

- Efficiency in the best case:
  n*log(n) calculation steps (comparisons)

- It can be shown:
  - On average, Quick Sort requires
    "in the order of n*log(n)" calculation steps

2nd case study

```
; (listof number) -> (listof number)
; to create a list with the numbers of l in ascending order
(define (qsort l)
   (cond
      [(empty? l) empty]
      [else
       (append
       (qsort (smaller-or-equal-than (first l) (rest l)))
       (list (first l) )
       (qsort (greater-than (first l) (rest l))))]))
; Number (listof Number) -> (listof Number)
; generates a list of all elements of xs that are smaller or equal than x
(define (smaller-or-equal-than x xs)
   (filter (lambda (y) (<= y x)) xs))
; Number (listof Number) -> (listof Number)
; generates a list of all elements of xs that are greater than x
(define (greater-than x xs)
   (filter (lambda (y) (> y x)) xs))
```

Follows the structure of l

But: recursive call with generated value.

2nd case study

```
; (listof number) -> (listof number)
; to create a list with the numbers of l in ascending order
(define (qsort l)
  (cond
    [(empty? l) empty]
    [else
     (append
     (qsort (smaller
     (list (first l) )
     (qsort (greate
; Number (listof Nu
; generates a list of                                      han x
(define (small
  (filter (lambda (y)
; Number (listof Num
; generates a list of all elements of xs that are greater than x
(define (greater-than x xs)
  (filter (lambda (y) (> y x)) xs))
```

By the way: filter is a built-in higher-order function for list processing. Together with map and foldr, it belongs to the "standard list functions".

Design of generative recursive functions

- Recursion not on result of selector but on generated value
- Case differentiation does not necessarily follow the structure of the data
- General

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
     ... problem ...
     (generative-recursive-fun (generate-problem-1 problem))
     ...
     (generative-recursive-fun (generate-problem-n problem))))))
```

Recursion termination

Recursive call

Generation of sub-problems

Design of generative recursive functions

1. What is a trivially solvable problem?
2. What is the solution to a trivially solvable problem?
3. How do we generate new problems that are easier to solve than the original problem? How many new problems should we generate?
4. How do we calculate the solution of the original problem from the solutions of the generated problems? Do we need the original problem (or a part of it) again?
5. Does the algorithm terminate?

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	
2. trivial solution	
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	Ball has already collided.
2. trivial solution	
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	Ball has already collided.
2. trivial solution	Current position
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	Ball has already collided.
2. trivial solution	Current position
3. generation of new problem	Calculate movement step
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	Ball has already collided.
2. trivial solution	Current position
3. generation of new problem	Calculate movement step
4. calculate the solution	Partial solution is already a complete solution
5. scheduling	

Design of generative recursive functions - 1st example

- Problem: When does the ball collide?

1. trivial problem:	Ball has already collided.
2. trivial solution	Current position
3. generation of new problem	Calculate movement step
4. calculate the solution	Partial solution is already a complete solution
5. scheduling	*(see later)*

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	
2. trivial solution	
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Philipps Universität Marburg

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	Sorting an empty list
2. trivial solution	
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	Sorting an empty list
2. trivial solution	The empty list
3. generation of new problem	
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	Sorting an empty list
2. trivial solution	The empty list
3. generation of new problem	Select a pivot element, generate two sub-lists
4. calculate the solution	
5. scheduling	

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	Sorting an empty list
2. trivial solution	The empty list
3. generation of new problem	Select a pivot element, generate two sub-lists
4. calculate the solution	Merge the sorted sub-lists with pivot element in the middle
5. scheduling	

Philipps Universität Marburg

Design of generative recursive functions - 2nd example

- Problem: Quick Sort

1. trivial problem:	Sorting an empty list
2. trivial solution	The empty list
3. generation of new problem	Select a pivot element, generate two sub-lists
4. calculate the solution	Merge the sorted sub-lists with pivot element in the middle
5. scheduling	*(see later)*

Use of generative recursion

- Typical situations
- Input has recursive data structure
  - Structural recursion is ...
    - ... not possible
      e.g. the solution to the original problem cannot be calculated from the result of the recursion
    - ... too complicated
    - ... too inefficient
- Input is not recursive
  - Number of calculation steps not proportional to the size of the input

Use of generative recursion

```
(define (generative-recursive-fun
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
     ... problem ...
     (generative-recursive-fun (generate-problem-1 problem))
     ...
     (generative-recursive-fun (generate-problem-n problem))))))
```

Question 1

Question 2

Question 4

Question 3

Use of generative recursion

1. trivial problem:
2. trivial solution
3. generation of new problem
4. calculate the solution

- Tests for
  - Trivial case
  - Recursive case
- Structural recursion
  - Answers to questions 1 and 3 already anchored in the template
  - Answers to questions 2 and 4 follow almost automatically
- Generative recursion
  - More creativity required

Scheduling

- Structural recursion always terminates:
  - Data structure is finite
  - input becomes smaller with each recursive call
  - Base case must be achieved

- With generative recursion
  - No fixed relationship between the size of the input and the size of the sub-problems
  - Scheduling must be shown individually

Scheduling - example

```
(define (qsort l)
  (cond
    [(empty? l) empty]
    [else
     (append
     (qsort (smaller-or-equal-than (first l) l))
     (list (first l))
     (qsort (greater-than (first l) (rest l))))]))
```

Error: Instead of searching in "rest" of the input, search in entire list (also contains pivot element)

```
>(qsort (list 5))
```

The result?

(smaller-or-equal-than 5 (list 5))
→ (list 5)
Identical to input, therefore endless recursion.

Show scheduling

1. Definition of a mapping
   - Theorem of function arguments → Natural number
   - Result value corresponds to the size of the problem from the algorithm's point of view
2. Show that the size of the input decreases strictly with recursive calls

- Size corresponds to the (maximum) recursion depth

Show scheduling

- Quick Sort
  - Size of the input: Length of the list (input argument)
  - Original input
    - (cons x xs)
    - Size: n
    - Then the size of xs is: n - 1
  - Help functions return a partial list
    - (smaller-or-equal-than x xs)
    - (greater-than x xs)
    - → Input size in all recursive calls <= n - 1
  - Input size is strictly smaller

Show scheduling

- First Collision
  - Size of the input: ?

We cannot find an image.

But maybe there is one. We cannot prove "non-termination" in this way.

In this case, however, first-collision does not always terminate.

Functions as values: Closures

- The previous program is correct!
- But:
  - Not one function is returned
  - But a "closure"
- Closure
  - Combination of function definition and
  - A local environment
- A closure is a so-called "functional closure"

Functions as values: Closures

- Using a closure like a function
- Closure call with arguments possible
- When evaluating the closure, the definitions from the bound local environment are used

```
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
  g))
(local [(define f (derivative exp))
  (define delta-x 10000)]
  (f 5))
```

Not used in the evaluation of f.

Lambdas

- Locally defined functions are often only used once
- Is it worth the effort?
  - Assigning a name
  - Elaborate syntax

```
(local [(define (double x) (* 2 x))]
  (map double lon))
```

Lambdas

- "Anonymous" functions: "lambdas"
- Are defined directly at the point where they are used
- `(map (lambda (x) (* 2 x)) lon)`

Lambdas

- General form:
  - (lambda (x_1 ... x_n) exp)
  - Corresponds to: (local [(define (f x-1 ... x-n) exp)] f)

- Abbreviation: Use of the Greek letter λ

  (map (λ (x) (* 2 x)) lon)

- Lambdas and local functions are not equivalent:
  - Lambdas have no name
  - They can therefore also not be called up recursively

Language simplification with lambdas

- However, lambdas together with constant definitions are a substitute for function definitions
  - (define (f x-1 ... x-n) exp) $\xrightarrow{\text{Transf.}}$ mbda (x-1 ... x-n) exp))
- So far
  - Function calls either begin with a function name
  - Or have the form $(exp_0\ exp_1\ \ldots\ exp\ )_n$
    - Where exp_0 is evaluated for a function
    - More precisely, exp_0 is evaluated for a closure
- With lambdas and constants
  - In a function call, exp_0 can be the name of a constant with a closure as the value

Lambda calculation

- Lambda calculation
  - Minimal model of programming languages
  - Fully described by formal rules
- Idea
  - All language constructs are syntactic sugar and can be transformed into lambda expressions
  - Properties that can be proven using the lambda calculus then also apply to these language constructs
- Language constructs that can be realized using lambda expressions, e.g:
  - Figures
  - Truth values
  - Constant definitions
  - Lists
  - Structures