

Introduction to Computer Science 1

Topic 8: Accumulating Knowledge

Prof. Bernt Schiele
Dr. Guido Rößling



Solution using structural recursion



```
;; relative-2-absolute : (listof number) -> (listof number)
;; to convert a list of relative distances to a
;; list of absolute distances the first item in the list
;; represents the distance to the origin
(define (relative-2-absolute alon)
  (cond
    [(empty? alon) empty]
    [else (cons (first alon)
                (add-to-each
                 (first alon)
                 (relative-2-absolute (rest alon))))]))

;; add-to-each : number (listof number) -> (listof number)
;; to add n to each number in alon
(define (add-to-each n alon) ...)
```

4

Back to the subject ...



- What is the time complexity of `relative-2-absolute` in terms of the length of the list?
- Let $T(n)$ be the time complexity of `relative-2-absolute`, and $U(n)$ the time complexity of `add-to-each`
- $U(n) = 1 + U(n-1) \rightarrow U(n) = n \in O(n)$
- $T(n) = 1 + T(n-1) + U(n-1)$
 $= 1 + T(n-1) + n-1$
 $= n + T(n-1)$
 $= n + (n-1) + \dots + 1$
 $= n*(n+1) / 2 = (n^2+n)/2$
 $\in O(n^2)$
- We can derive the time complexity of the version using `foldr`, `map` from the time complexity of `foldr`, `map`
 - `foldr f base alox` $\rightarrow T(n) = n * \Theta(n)$, where n is the length of `alox`, and $\Theta(n)$ is the time complexity of `f`

7

Recursive functions

- All recursive functions we have seen so far are context free
 - They cope with a sub problem without knowing the whole problem
- Advantage: Such functions are easy to develop, maintain, etc.
- Disadvantage: Some functions get inefficient or complicated

2

Solution using structural recursion



```
;; add-to-each : number (listof number) -> (listof number)
;; to add n to each number in alon
(define (add-to-each n alon)
  (cond
    [(empty? alon) empty]
    [else (cons (+ (first alon) n)
                (add-to-each n (rest alon))))]))
```

- Before discussing the disadvantages, let's apply our knowledge about higher-order functions
 - `(add-to-each n alon) -> (map (lambda (m) (+ n m)) alon)`
 - `(relative-2-absolute alon) -> (foldr (lambda (x xs) (cons x (add-to-each x xs))) empty alon)`
- Note: the function that is getting folded is not associative
 - using `foldl` instead of `foldr` yields a different result!
- How did we find the last equation?

5

What is the problem?

- The problem is that `relative-2-absolute` is context free
 - The recursive call for L in the list $(\text{cons } N \ L)$ is doing exactly the same as for another list $(\text{cons } K \ L)$
- Idea: Another parameter accumulates the knowledge about the context, i.e., the accumulated distance in this case

```
(define (rel-2-abs alon accu-dist)
  (cond
    [(empty? alon) empty]
    [else (cons (+ (first alon) accu-dist)
                (rel-2-abs
                 (rest alon)
                 (+ (first alon) accu-dist))))]))
```

8

Example

- Given: A list of points with the relative distances between two successive points



- Task: Calculate the absolute distances from the origin to each point



3

Reminder: foldr



```
;; foldr : (X Y -> Y) Y (listof X) -> Y
;; (foldr f base (list x-1 ... x-n)) =
;; (f x-1 ... (f x-n base))
(define (foldr f base alox)
  (cond
    [(empty? alox) base]
    [else (f (first alox) (foldr f base (rest alox)))]))
```

How can we specialize `foldr` to get `relative-2-absolute`?

```
(define (relative-2-absolute alon)
  (cond
    [(empty? alon) empty]
    [else (cons (first alon)
                (add-to-each
                 (first alon)
                 (relative-2-absolute (rest alon))))]))
```

- `f = (lambda (x xs) (cons x (add-to-each x xs)))`

6

Solution

- Function is not yet equal to `relative-2-absolute`
 - an additional parameter
 - the parameter has an initial value of 0

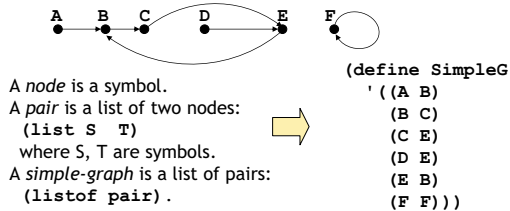
→

```
(define (relative-2-absolute2 alon)
  (local
    ((define (rel-2-abs alon accu-dist)
      (cond
        [(empty? alon) empty]
        [else (cons (+ (first alon) accu-dist)
                    (rel-2-abs
                     (rest alon)
                     (+ (first alon) accu-dist)))])))
    (rel-2-abs alon 0)))
```

9

Example with generative recursion

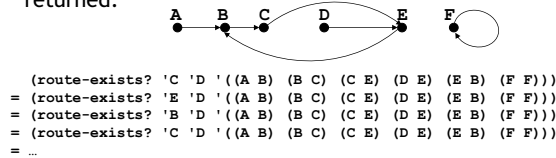
- The problem to accumulate knowledge in certain situations is not unique to structural recursion, but also exists in the case of generative recursion
- Example: Find paths in a simple graph, in which each node has a single outgoing edge only (cycles allowed!)



10

Cause of the fault

- Let's consider a scenario, in which "false" has to be returned:



- After some calls, the function calls itself with the same parameters
 - Infinite loop
- Cause: The function "forgets", with which parameters it has already been called

13

Preliminary summary

- In the case of both, structural and generative recursion, an accumulator is sometimes needed
 - in an example with structural recursion, we improved the runtime performance significantly
 - the example that used generative recursion it didn't work without an accumulator
- Now, let's consider in which cases we need an accumulator in general!

16

Example with generative recursion

- The header is simple:


```
;; route-exists? : node node simple-graph -> boolean
;; to determine whether there is a route from
;; orig to dest in sg
(define (route-exists? orig dest sg) ...)
```
- For the implementation of the body, we need to answer four basic questions of generative recursion:
 - What is a trivially solvable problem?
 - The problem is trivial if the nodes orig and dest are the same.
 - What is a corresponding solution?
 - Easy: true.
 - How do we generate new problems?
 - If orig is not the same as dest, there is only one thing we can do, namely, go to the node to which orig is connected and determine whether a route exists between it and dest.
 - How do we relate the solutions?
 - There is no need to do anything after we find the solution to the new problem. If orig's neighbor is connected to dest, then so is orig.

11

Solution

- Accumulation of nodes already visited

```
;; route-exists-accu? :
;; node node simple-graph (listof node) -> boolean
;; to determine whether there is a route from orig to dest
;; in sg, assuming the nodes in accu-seen have
;; already been inspected and failed to deliver a solution
(define (route-exists-accu? orig dest sg accu-seen)
  (cond
    [(symbol=? orig dest) true]
    [else (route-exists-accu? (neighbor orig sg) dest sg
                              (cons orig accu-seen))]))
```

- The accumulation on its own is not enough; we also have to use the accumulated knowledge!

14

Designing functions with accumulators

- In many cases, one recognizes the need for an accumulator after a first version of a function has been implemented
- The keys to the development of an accumulator-style function are:
 - to recognize that the function benefits from, or needs, an accumulator;
 - to understand what the accumulator represents.

17

Now, the solution is simple...

```
;; route-exists? : node node simple-graph -> boolean
;; to determine whether there is a route from orig to dest in sg
(define (route-exists? orig dest sg)
  (cond
    [(symbol=? orig dest) true]
    [else (route-exists? (neighbor orig sg) dest sg)]))

;; neighbor : node simple-graph -> node
;; to determine the node that is connected to a-node in sg
(define (neighbor a-node sg)
  (cond
    [(empty? sg) (error "neighbor: impossible")]
    [else (cond
            [(symbol=? (first (first sg)) a-node)
             (second (first sg))]
            [else (neighbor a-node (rest sg))]))]))
```

- ...but it doesn't work!
 - route-exists? can never return false

12

Solution

- Add a check, specialize...

```
;; route-exists2? : node node simple-graph -> boolean
;; determine whether there is a route from orig to dest in sg
(define (route-exists2? orig dest sg)
  (local ((define (re-accu? orig dest sg accu-seen)
          (cond
            [(symbol=? orig dest) true]
            [(contains? orig accu-seen) false]
            [else (re-accu?
                        (neighbor orig sg)
                        dest
                        sg
                        (cons orig accu-seen))]))))
    (re-accu? orig dest sg empty)))
```

15

Recognizing the need for an accumulator

- We have seen two reasons
 - Performance
 - Function does not work correctly without accumulator
- They are the most prevalent reasons for adding accumulator parameters.
- In either case, it is critical that we first build a complete function *based on a design recipe*.
- Then we study the function and look for one of the following characteristics:

18

Potential candidate for an accumulator

- Heuristic: If the function is structurally recursive and if the result of a recursive application is processed by an auxiliary, recursive function, then we should consider the use of an accumulator parameter.
- Example

```
;; invert : (listof X) -> (listof X)
;; to construct the reverse of alox
;; structural recursion
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item (first alox) (invert (rest alox)))]))

;; make-last-item : X (listof X) -> (listof X)
;; to add an-x to the end of alox
;; structural recursion
(define (make-last-item an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (make-last-item an-x (rest alox)))]))
```



Analysis

- Clearly, invert cannot forget anything, because it only reverses the order of items in the list. Hence we might just wish to accumulate all items that rev encounters. This means
 - that accumulator stands for a list, and
 - that it stands for all those items in alox0 that precede the alox argument of rev.
 - the initial value for accumulator is empty.
 - When rev recurs, it has just encountered one extra item: (first alox). To remember it, we can cons it onto the current value of accumulator.

22

[Direct Implementation of invert]

- Variant without accumulator...

```
;; invert : (listof X) -> (listof X)
;; to construct the reverse of alox
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else
     (append (invert (rest alox)) (list (first alox)))]))
```

25

Accumulator-style functions

- If we decide to write a function in accumulator-style, we introduce the accumulator in two steps
 1. Definition and preparation of the accumulator
 - Which knowledge about the parameters need to be stored in the accumulator?
 - For instance, for the function relative-2-absolute it was sufficient to accumulate the whole distance so far, i.e., the accumulator was a single number
 - For the routing problem, we had to store each node that has already been visited. Thus, the accumulator was a list nodes.
 2. Using the accumulator

20

Extended version of invert

```
;; invert : (listof X) -> (listof X)
;; to construct the reverse of alox
(define (invert alox0)
  (local ;; accumulator is the reversed list of all those items
        ;; in alox0 that precede alox
        (define (rev alox accumulator)
          (cond
            [(empty? alox) ...]
            [else
             ... (rev (rest alox) (cons (first alox) accumulator))
             ...]))
    (rev alox0 empty)))
```

- accumulator is not just the items in alox0 that precede but a list of these items in reverse order.

23

Definition of accumulator invariants

- Key step: precise description of the role of accumulator. In general, an ACCUMULATOR INVARIANT describes a relationship between
 - the argument(s) proper of the function,
 - the current argument of the auxiliary function, and
 - the accumulator
- Let's take a look at the definition of accumulator invariants in another example that does not need an accumulator
 - This enables us to compare both variants
- Example: Summation of numbers in a list

```
;; sum : (listof number) -> number
;; to compute the sum of the numbers in alon
;; structural recursion
(define (sum alon)
  (cond
    [(empty? alon) 0]
    [else (+ (first alon) (sum (rest alon)))]))
```

26



Definition and preparation of the accumulator

- The best way to discuss the accumulation process is to introduce a template of the accumulator-style function via a local definition and to name the parameters of the function differently from those of the auxiliary function.
- Example: transforming the function invert into accumulator-style

```
;; invert : (listof X) -> (listof X)
;; to construct the reverse of alox
(define (invert alox0)
  (local ;; accumulator ...
        (define (rev alox accumulator)
          (cond
            [(empty? alox) ...]
            [else
             ... (rev (rest alox)
                     (cons (first alox) accumulator))
             ... ])))
    (rev alox0 ...)))
```

21

Using the accumulator

- If accumulator is the list of all items in alox0 that precede alox in reverse order, then, if alox is empty, accumulator stands for the reverse of alox0.

```
;; invert : (listof X) -> (listof X)
;; to construct the reverse of alox
(define (invert alox0)
  (local ;; accumulator is reversed list of all those items
        ;; in alox0 that precede alox
        (define (rev alox accumulator)
          (cond
            [(empty? alox) accumulator]
            [else
             (rev (rest alox) (cons (first alox) accumulator))]))
    (rev alox0 empty)))
```

24

Definition of accumulator invariants

- First step towards an accumulator variant: Template

```
;; sum : (listof number) -> number
;; to compute the sum of the numbers in alon0
(define (sum alon0)
  (local ;; accumulator ...
        ;;
        (define (sum-a alon accumulator)
          (cond
            [(empty? alon) ...]
            [else
             ... (sum-a (rest alon)
                       ... (first alon) ... accumulator)
             ... ])))
    (sum-a alon0 ...)))
```

- The goal of sum is the summation of numbers
- Obvious accumulator invariant: the accumulator represents the sum of all numbers that have already been processed

27

Definition of accumulator invariants

- First step towards an accumulator variant: Template

```
;; sum : (listof number) -> number
;; to compute the sum of the numbers in alon0
(define (sum alon0)
  (local ( ;; accumulator is the sum of the numbers that
          ;; preceded those in alon in alon0
          (define (sum-a alon accumulator)
            (cond
              [(empty? alon) ...]
              [else
               ... (sum-a (rest alon)
                         (+ (first alon) accumulator))
               ... ])))
    (sum-a alon0 0)))
```

28

Summary

- Some functions can only be implemented correctly using the accumulator-style
 - Example: `route-exists?`
- Sometimes, the use of accumulator-style functions can yield performance improvements
 - Example: `relative-2-absolute`
- However, accumulator-style functions are not *always* faster
 - Example: `sum`
- We will see later that accumulator-style functions are very similar to loops in programming languages such as Java, Pascal, etc.

31

Definition of accumulator invariants

- Now, the remainder is simple...

```
;; sum : (listof number) -> number
;; to compute the sum of the numbers in alon0
(define (sum alon0)
  (local ( ;; accumulator is the sum of the numbers
          ;; that preceded those in alon in alon0
          (define (sum-a alon accumulator)
            (cond
              [(empty? alon) accumulator]
              [else (sum-a (rest alon)
                           (+ (first alon) accumulator))]))
    (sum-a alon0 0)))
```

- The key is a precise definition of the accumulator invariant; then, the remainder is straightforward

29

Comparison

- Original version

```
(sum (list 10.23 4.50 5.27))
= (+ 10.23 (sum (list 4.50 5.27)))
= (+ 10.23 (+ 4.50 (sum (list 5.27))))
= (+ 10.23 (+ 4.50 (+ 5.27 (sum empty))))
= (+ 10.23 (+ 4.50 (+ 5.27 0)))
= (+ 10.23 (+ 4.50 5.27))
= (+ 10.23 9.77)
= 20.0
```

- Accumulator version

```
(sum (list 10.23 4.50 5.27))
= (sum-a (list 10.23 4.50 5.27) 0)
= (sum-a (list 4.50 5.27) 10.23)
= (sum-a (list 5.27) 14.73)
= (sum-a empty 20.0)
= 20.0
```

30