Introduction to Programming and its Mathematical Foundations

Lesson 8: Lists, Part 2 (Lesson 3 forms Lists, Part 1)

Notation: A *list* in functional programming is commonly written as a sequence of elements, all of the same type e.g., `[2,18,9, 10]`. The *component type* can be any simple or compound type that is legal in Gofer, e.g., `Int`, and the aggregate *list type* is written as the component type in brackets, e.g., `[Int]`

This notation is "syntactic sugarcoating" that hides the true, recursive, nature of the list structure (see Figures 1 below).

(Definition): A *list* is either empty [ ], or consists of a *head* (single element) and a *tail* (list of elements satisfying this recursive definition), all elements of the same type. The head and tail are connected together by the `cons` operator, written `::` in Gofer. (Note: In standard Haskell notation, `cons` is written with a single colon, `:`)

(Recursive) A non-empty tail has a head (single element(and a tail (itself another list), till finally the tail is the empty list [ ]. The head and tail are connected together by the cons operator, written `::` , so `[2,18,9,10]` is identically equal to `2::(18::(9::(10::[])))` , as illustrated in Figures 1a and 1b. That is, the value of `[2,18,9,10] == 2::(18::(9::(10::[])))` is `True`.

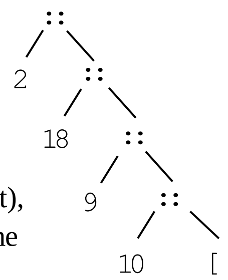

Figure 1a: Structure of [2, 18, 9, 10]

Lists can be represented in a natural way as lopsided trees, see Figure 1. The inverse operators of `::` are the built-in unary functions `head` and `tail`, and there is a `length` function, so that `length([2, 18, 9, 10])` is `4`, and `length([])` is `0`.

Further, just like we have the `n+1` pattern for positive integers, there is an `x::xs` pattern for nonempty lists. There is no particular significance to the names—`x` stands for one element, and `xs` stands for many elements. Just as the base case for the `n+1` pattern is `0`, for lists it is `[]`. Note that neither the `n+1` pattern nor the `[` pattern can match the base case—the `n+1` pattern matches integer arguments 1 and above, while the `x::xs` pattern matches non-empty lists, i.e., lists of size 1 or more.

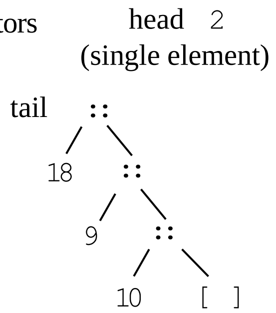

Figure 1b: the head is `2` and the tail is `[18, 9, 10]`

Signatures	*Examples*	*Application*
`head: [a] -> a`	`head.[2,18,9,0] = 2`	`head.(x::xs) = x`
`tail: [a] -> [a]`	`tail.[2,18,9,0] = [18,9,0]`	`tail.(x::xs) = xs`
`length: [a] -> Int`	`length.[2,18,9,0] = 4`	`length.[] = 0`

The various forms of recursive function definition for integers, as shown in file `recfuncs.gs`, can be used in recursive function definitions for lists. Assume that `s` is a list, `x::xs` is a list pattern. Just as the induction case for integers connects parameter `n` to `n-1`, we have `s` connected to `tail.s` on the right hand side, and as the induction case connects pattern `n+1` to `n`, we have `x::xs` connected to `x`. To practically observe these "connections," inspect the details of the example for `len`, given below

Assume `len` is a function that finds list length. (The name of the built-in function is not len but `length` .) The different versions of `len` are structured as below. The first uses the `x::xs` pattern, the second uses function tail, and the third uses a conditional expression. The `n+1` pattern matches integers `1` and above, the `x::xs` pattern matches non-empty lists (of length `>= 1`).

Induction case first (using pattern `x::xs`)

```
len.(x::xs) = 1+len. xs
len.[] = 0
```

Base case first (using parameter `s`)

```
len.[] = 0
len. s  = 1 + len.(tail.s)
```

Conditional expression: `len. s = if s==[] then 0 else 1 + len.(tail.s)`

♋ ♦ ♋

PROOF BY STRUCTURAL INDUCTION, FOR LISTS

Remark: Structural induction, a generalisation of mathematical induction, is used to prove that some proposition `P(x)` holds for all instances `x` of some "recursively defined" structure, such as lists or trees. Often, such a proof proceeds by natural induction on the size of the structure.

Let `P(s)` be a well-formed proposition that involves `s`, a list in the sense of purely functional programming. We want to show that `P(s)` holds, i.e., `P(s)` is true, for all lists `s` in a given context, e.g., a certain list type.

The base case will be for small-sized lists, typically the empty list `[]`, and sometimes the singleton list `[x]`. As with natural numbers, the base case for lists is usually shown to be correct by inspection. The induction case will be for non-empty lists, denoted by the pattern `x::xs`, or lists with >2 elements `x::y::ys`. The induction hypothesis is that `P(xs)` is true, and the induction step is to do "something" with x, the list head, and the value of the recursive call on the list tail `xs`, and conclude that `P(x::xs)` holds, i.e., show that `P(xs)=>P(x::xs)`

```
len: [a] -> Int
len . [ ] = 0
len . (x::xs) = 1 + len . xs
```

`sum: Num.a => [a] -> Int`–all numeric types `a`
```
sum . [ ] = 0
```
`sum . (x::xs) = x + sum . xs`Below, you will find descriptions of problems that can be solved naively (i.e., from first principles, i.e., using the "base case first" or "induction case first" or "conditional expression" approaches, (according to your thinking or problem-solving strategy.)

To get a complete expression out of the "mention" below, you have to supply the list argument, e.g., when the argument is list `s`, the complete expressions are, e.g., `len.s` or `oll.p.s` or `isin.q.s`.

Function	mention	description
`len`	`len`	length of a list (same as builtin `length`)
`total`	`total`	sum of list elements (same as builtin `sum`)
`produc`	`produc`	product of list elements (same as `product`)
`countevens`	`countevens`	count the even numbers in the list
`sumEvens`	`sumEvens`	sum the even numbers in the list
`sumalt1`	`sumalt1`	sum of alternating elements starting from the head of the list
`sumalt2`	`sumalt2`	sum of alt elements, not counting the head, i.e., `sumalt1` of the tail
`anee`	`anee.p`	does any element satisfy predicate `p`, same as builtin `any`
`oll`	`oll.p`	do all list elements satisfy predicate `p`, same as builtin `all`
`isin`	`isin.q`	does `q` occur in the list
`allare`	`allare.q`	do all the list elements have the value `q`

♋ ♦ ♋