Concepts of Higher Programming Languages
Recursive Functions

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Introduction

Many **functions can naturally be defined in terms of other functions**.

```
fact :: Int -> Int
fact n = product [1..n]
```

`fact` maps any integer `n` to the product of the integers between `1` and `n`.

Introduction

Expressions are **evaluated** by a stepwise process of applying functions to their arguments.

```
  fact 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

Recursive Functions

Definition

In Haskell, **functions can also be defined in terms of themselves**. Such functions are called **recursive**.

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Base Case

Recursive Case

`fact` maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

Recursive Functions

Evaluating `fact`

```
  fact 3
=
  3 * fact 2
=
  3 * (2 * fact 1)
=
  3 * (2 * (1 * fact 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Recursive Functions

- `fact 0 = 1` is appropriate because 1 is the **identity** for **multiplication**: `1 * x = x = x * 1`
- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> fact (-1)
*** Exception: stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are **simpler** to define in terms of other functions.
- As we shall see, however, many functions can **naturally** be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of **induction**.

Recursion on Lists

Recursion is not restricted to numbers, and can be used to define functions on **lists**.

```
product :: Num a => [a] -> a
product []     = 1
product (n:ns) = n * product ns
```

`product` maps the **empty** list to `1`, and any **non-empty** list to its head multiplied by the product of its tail.

Recursion on Lists

Evaluating product

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Recursion on Lists

Using the same pattern of recursion as in `product` we can define the `length` function.

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

`length` maps the **empty** list to 0, and any **non-empty** list to the successor of the length of its tail.

Recursion on Lists

Evaluating `length`

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Recursion on Lists

Using a similar pattern of recursion we can define the **reverse** function on lists.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

`reverse` maps the **empty** list to the empty list, and any **non-empty** list to the `reverse` of its tail appended to its head.

Recursion on Lists

Evaluating `reverse`

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Recursion on Lists

Problem of **reverse**: it is slow.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- It is **not tail-recursive**.
- (++) for each element!

Can we do better?

Recursion on Lists

A **fast** `reverse`:

```
fastRev :: [a] -> [a]
fastRev xs = fastRev' xs []
  where
    fastRev' :: [a] -> [a] -> [a]
    fastRev' [] acc     = acc
    fastRev' (x:xs) acc = fastRev' xs (x:acc)
```

✓It is tail-recursive
✓Avoids (++) for each element, uses cons (:)

Recursion on Lists

Testing timing in GHCi
Turn on timing with `:set s+` command

Reversing a list with 10 million elements

```
ghci Reverse.hs

Prelude> :set +s
Prelude> head $ reverse [1..10000000]
(4.74 secs, 2,904,959,424 bytes)
Prelude> head $ fastRev [1..10000000]
(3.15 secs, 1,840,070,248 bytes)
```

If you try this you should use the implementation of `reverse` **as shown in the slides**. The Prelude implementation of `reverse` is actually **even faster** than `fastRev` with a performance of `(1.03 secs, 1,120,070,128 bytes)`.

Multiple Arguments

Functions with **more than one argument** can also be defined using **recursion**.

Zipping the elements of two lists:

```
zip :: [a] -> [b] -> [(a,b)]
zip ?
zip ?
zip ?
```

Multiple Arguments

Functions with **more than one argument** can also be defined using **recursion**.

Zipping the elements of two lists:

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Multiple Arguments

Remove the first n elements from a list:

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

Appending two lists:

```
(++) :: [a] -> [a] -> [a]
(++) [] ys     = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

Multiple Arguments

Remove the first n elements from a list:

```
drop :: Int -> [a] -> [a]
drop ?
drop ?
drop ?
```

Appending two lists:

```
(++) :: [a] -> [a] -> [a]
(++) ?
(++) ?
```

Mutual Recursion

We can define recursion also **mutually** with more than one function. For example testing for even / odd:

```
even :: Int -> Bool
even ?
even ?

odd :: Int -> Bool
odd ?
odd ?
```

Mutual Recursion

We can define recursion also **mutually** with more than one function. For example testing for even / odd:

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

Quicksort

The **Quicksort** algorithm for sorting a list of values can be specified by the following two rules:

1. The empty list is already sorted.

2. Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $\geq$ the head, and then appending the resulting lists on either side of the head value.

Quicksort

Using recursion, this specification can be translated directly into an implementation:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

This is probably the simplest implementation of quicksort in any programming language!

Quicksort

For example (abbreviating qsort as q):

```
                    q [3,2,4,1,5]

              q [2,1] ++ [3] ++ q [4,5]

      q [1] ++ [2] ++ q []        q [] ++ [4] ++ q [5]

      [1]             []          []           [5]
```