Recursion

dannymaate

12 July 2022

1 Recursion

When pattern matching on `num` types you must include the equality class since `numbers` are polymorphic. Haskell won't know if, say 0, means `(0::Int)`, `(0::Integer)`, `(0::Double)`, ...
Note to check if an input is less than zero, the *type has to be an instance of* `Ord`.

Primitive Recursion*

A function `f :: Integer -> Integer` is defined by *primitive recursion* if

1. some *base case* *f(0)* is given outright (e.g. no more calls to any function), **and**
2. *f(n)* (for n > 0) is defined in terms of n and *f(n - 1)*, *e.g. factorial, power*.

Linear Recursive* A function definition where each execution *calls itself at most once*.

Tail Recursion*

A function is *tail recursive* or *iterative* if each execution of the function either

1. is a *base case* that has no more calls to any functions, **or**
2. the function *calls itself only* with different values for arguments

Note: the last function call should be to itself.

Iteration Invariant A property of the algorithm that must be true regardless of the winding phase, e.g. the *iteration invariant* for the tail recursive fact is `h_fact ans n = ans.n!`

```
h_fact ans 0 = ans
h_fact ans n = h_fact (ans*n) * (n - 1)
```

Termination Checking To prove a tail recursive algorithm *terminates* the *bound value* must be

1. always *non-negative* and
2. *decreasing* at reach recursive call

2 Recursion: Advice

Define the type `init :: [a] -> [a]`

Enumerate the cases

```
init [x] = ?
init (x:xs) = ?
```

Define base cases `init [x] = []`

Define inductive cases `init (x:xs) = [x] ++ (init xs)`

* excerpt originally from lecture material.