Exploring Languages with Interpreters and Functional Programming
Chapter 26

H. Conrad Cunningham

2 August 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of August 2018 is a recent version of Firefox from Mozilla.

26 Program Synthesis

26.1 Chapter Introduction

The previous chapter illustrated how to state and prove Haskell "laws" about already defined functions.

This chapter illustrates how to use similar reasoning methods to synthesize (i.e. derive or calculate) function definitions from their specifications.

The next chapter applies these program synthesis techniques to a larger set of examples on text processing.

26.2 Motivation

This chapter deals with program synthesis.

In the *proof* of a property, we take an existing program and then demonstrate that it satisfies some property.

In the *synthesis* of a program, we take a property called a *specification* and then synthesize a program that satisfies it [Bird 1988]. (Program synthesis is called program *derivation* in other contexts, such as in the Gries textbook [Gries 1981].)

Both proof and synthesis require essentially the same reasoning. Often a proof can be turned into a synthesis by simply reversing a few of the steps, and vice versa.

26.3 Fast Fibonacci Function

This section is based on Sections 5.4.5 and 5.5 of the [Bird 1988] and Section 4.5 of [Hoogerwoord 1989].

A (second-order) Fibonacci sequence is the sequence in which the first two elements are 0 and 1 and each successive element is the sum of the two immediately preceding elements:

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

As we have seen in Chapter 9, we can take the above informal description and define a function to compute the `n`th element of the Fibonacci sequence. The definition is straightforward. Unfortunately, this algorithm is quite inefficient, `O(fib n)`.

```
fib :: Int -> Int
fib 0          = 0
```

```
fib 1           = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

In Chapter 9 we also developed a more efficient, but less straightforward, version by using two accumulating parameters. This definition seemed to be "pulled out of thin air". Can we synthesize a definition that uses the more efficient algorithm from the simpler definition above?

Yes, but we use a slightly different approach than we did before. We can improve the performance of the Fibonacci computation by using a technique called *tupling* [Hoogerwoord 1989] as we saw in Chapter 20.

The tupling technique can be applied to a set of functions with the same domain and the same recursive pattern. First, we define a new function whose value is a tuple, the components of which are the values of the original functions. Then, we proceed to calculate a recursive definition for the new function.

This technique is similar to the technique of adding accumulating parameters to define a new function.

Given the definition of `fib` above, we begin with the specification [Bird 1988]:

```
twofib n = (fib n, fib (n+1))
```

and synthesize a recursive definition by using induction on the natural number `n`.

Base case `n = 0`:

```
twofib 0
```

= { specification }

```
(fib 0, fib (0+1))
```

= { arithmetic, `fib.1, fib.2` }

```
(0,1)
```

This gives us a definition for the base case.

Inductive case `n = m+1`:

Given that there is a definition for `twofib m` that satisfies the specification

```
twofib m = (fib m, fib (m+1))
```

calculate a definition for `twofib (m+1)` that satisfies the specification.

```
twofib (m+1)
```

= { specification }

```
(fib (m+1), fib ((m+1)+1))
```

= { arithmetic, `fib.3` }

```
    (fib (m+1), fib m + fib (m+1))
```

= { abstraction }

```
    (b,a+b)
    where (a,b) = (fib m, fib (m+1))
```

= { induction hypothesis }

```
    (b,a+b)
    where (a,b) = twofib m
```

This gives us a definition for the inductive case.

Bringing the cases together and rewriting `twofib (m+1)` to get a valid pattern, we synthesize the following definition:

```
twofib :: Int -> (Int,Int)
twofib 0         = (0,1)
twofib n | n > 0 = (b,a+b)
                   where (a,b) = twofib (n-1)

fastfib :: Int -> Int
fastfib n = fst (twofib n)
```

Above `fst` is the standard prelude function to extract the first component of a pair (i.e. a 2-tuple).

The key to the performance improvement is solving a "harder" problem: computing `fib n` and `fib (n+1)` at the same time. This allows the values needed to be "passed forward" to the "next iteration".

In general, we can approach the synthesis of a function using the following method.

- Devise a specification for the function in terms of defined functions, data, etc.
- Assume the specification holds.
- Using proof techniques (as if proving the specification), calculate an appropriate definition for the function.
- As needed, break the synthesis calculation into cases motivated by the induction "proof" over an appropriate (well-founded) set (e.g. over natural numbers or finite lists). The inductive cases usually correspond to recursive legs of the definition.

26.4 Sequence of Fibonacci Numbers

Now let's consider a function to generate a list of the elements `fib 0` through `fib n` for some natural number `n`. A simple backward recursive definition follows:

```
allfibs :: Int -> [Int]
allfibs 0         = [0]                          -- allfibs.1
allfibs n | n > 0 = allfibs (n-1) ++ [fib n]     -- allfibs.2
```

Using `fastfib`, each `fib n` calculation is `O(n)`. Each `++` call is also `O(n)`. The `fib` and the `++` are "in sequence", so each call of `allfibs` is just `O(n)`. However, there are `O(n)` recursive calls of `allfibs`, so the overall complexity is `O(n^2)`.

We again attempt to improve the efficiency by tupling We begin with the following specification for `fibs`:

```
fibs n = (fib n, fib (n+1), allfibs n)
```

We already have definitions for the functions on the right-hand side, `fib` and `allfibs`. Our task now is to synthesize a definition for the left-hand side, `fibs`.

We proceed by induction on the natural number `n` and consider two cases.

Base case `n = 0`:

```
fibs 0
```

= { `fibs` specification }

```
(fib 0, fib (0+1), allfibs 0)
```

= { `fib.1, fib.2, allfibs.1` }

```
(0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`

Given that there is a definition for `fibs m` that satisfies the specification

```
fibs m = (fib m, fib (m+1), allfibs m)
```

calculate a definition for `fibs (m+1)` that satisfies the specification.

```
fibs (m+1)
```

= { `fibs` specification }

```
(fib (m+1), fib (m+2), allfibs (m+1))
```

= { `fib.3, allfibs.2` }

```
(fib (m+1), fib m + fib (m+1), allfibs m ++ [fib (m+1)])
```

= { abstraction }

```
(b,a+b,c++[b])
where (a,b,c) = (fib m, fib (m+1), allfibs m)
```

= { induction hypothesis }

```
    (b,a+b,c++[b])
    where (a,b,c) = fibs m
```

This gives us a definition for the inductive case.

Bringing the cases together, we get the following definitions:

```
fibs :: Int -> (Int,Int,[Int])
fibs 0         = (0,1,[0])
fibs n | n > 0 = (b,a+b,c++[b])
                 where (a,b,c) = fibs (n-1)

allfibs1 :: Int -> [Int]
allfibs1 n = thd3 (fibs n)
```

Above `thd3` is the standard prelude function to extract the third component of a 3-tuple.

We have eliminated the `O(n) fib` calculations, but still have an `O(n)` append (`++`) within each of the `O(n)` recursive calls of `fibs`. This program is better, but is still `O(n^2)`.

Note that in the `c ++ [b]` expression there is a single element on the right. Perhaps we could build this term backwards using cons, an `O(1)` operation, and then reverse the final result.

We again attempt to improve the efficiency by tupling. We begin with the following specification for `fibs`:

```
fibs' n = (fib n, fib (n+1), reverse (allfibs n))
```

For convenience in calculation, we replace `reverse` by its backward recursive equivalent `rev`.

```
rev :: [a] -> [a]
rev []     = []              -- rev.1
rev (x:xs) = rev xs ++ [x]   -- rev.2
```

We again proceed by induction on { ‘n} and consider two cases.

Base case `n = 0`:

```
    fibs' 0
```

= { `fibs'` specification }

```
    (fib 0, fib (0+1), rev (allfibs 0))
```

= { `fib.1`, `fib.2`, `allfibs.1` }

```
    (0,1, rev [0])
```

= { `rev.2` }

```
    (0,1, rev [] ++ [0])
```

= { `rev.1`, `append.1` }

```
(0,1,[0])
```

This gives us a definition for the base case.

Inductive case `n = m+1`:

Given that there is a definition for `fibs' m` that satisfies the specification

```
fibs' m = (fib m, fib (m+1), allfibs m)
```

calculate a definition for `fibs' (m+1)` that satisfies the specification.

```
fibs' (m+1)
```

= { `fibs'` specification }

```
(fib (m+1), fib (m+2), rev (allfibs (m+1)))
```

= {`fib.3`, `allfibs.2` }

```
(fib (m+1), fib m + fib (m+1), rev (allfibs m ++ [fib (m+1)]))
```

= { abstraction }

```
(b, a+b, rev (allfibs m ++ [b]))
where (a,b,c) = (fib m, fib (m+1), rev (allfibs m))
```

= { induction hypothesis }

```
(b, a+b, rev (allfibs m ++ [b]))
where (a,b,c) = fibs' m
```

= { `rev (xs ++ [x]) = x : rev xs`, Note 1 }

```
(b, a+b, b : rev (allfibs m))
where (a,b,c) = fibs' m
```

= { substitution }

```
(b, a+b, b:c)
where (a,b,c) = fibs' m
```

This gives us a definition for the inductive case.

Note 1: The proof of `rev (xs ++ [x]) = x : rev xs` is left as an exercise.

Bringing the cases together, we get the following definition:

```
fibs' :: Int -> (Int,Int,[Int])
fibs' 0         = (0,1,[0])
fibs' n | n > 0 = (b,a+b,b:c)
                  where (a,b,c) = fibs' n

allfibs2 :: Int -> [Int]
allfibs2 n = reverse (thd3 (fibs' n))
```

Function `fibs'` is `O(n)`. Hence, `allfibs2'` is `O(n)`.

Are further improvements possible?

Clearly, function `fibs'` must generate an element of the sequence for each integer in the range `[0..n]`. Thus no complexity order improvement is possible.

However, from our previous experience, we know that it should be possible to avoid doing a reverse by using a tail recursive auxiliary function to compute the Fibonacci sequence. The investigation of this possible improvement is left to the reader.

For an `O(log2(n))` algorithm to compute `fib n`, see section 5.2 of Kaldewaij's textbook on program derivation [Kaldewaij 1990].

26.5 Synthesis of `drop` from `take`

Suppose that we have the following definition for the list function `take`, but no definition for `drop`.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take' (n-1) xs
```

Further suppose that we wish to synthesize a definition for `drop` that satisfies the following specification for any natural number `n` and finite list `xs`.

```
take n xs ++ drop n xs = xs
```

We proved this as a property earlier, given definitions for both `take` and `drop`. The synthesis uses induction on both `n` and `xs` and the same cases we used in the proof.

Base case `n = 0`:

```
  xs
```

= { specification, substitution for this case }

```
  take 0 xs ++ drop 0 xs
```

= { `take.1` }

```
  [] ++ drop 0 xs
```

= { `++` identity }

```
  drop 0 xs
```

This gives the equation `drop 0 xs = xs`.

Base case `n = m+1`:

```
  []
```

= { specification, substitution for this case }

```
  take (m+1) [] ++ drop (m+1) []
```

= { `take.2` }

```
  [] ++ drop (m+1) []
```

= { `++` identity }

```
  drop (m+1) []
```

This gives the defining equation `drop (m+1) [] = []`. Since the value of the argument `(m+1)` is not used in the above calculation, we can generalize the definition to `drop _ [] = []`.

Inductive case `n = m+1, xs = (a:as)`:

Given that there is a definition for `drop m as` that satisfies the specification:

```
  take m as ++ drop m as = as
```

calculate an appropriate definition for `drop (m+1) (a:as)` that satisfies the specification.

```
  (a:as)
```

= { specification, substitution for this case }

```
  take (m+1) (a:as) ++ drop (m+1) (a:as)
```

= { `take.3` }

```
  (a:(take m as)) ++ drop (m+1) (a:as)
```

= { `append.2` }

```
  a:(take m as ++ drop (m+1) (a:as))
```

Hence, `a:(take m as ++ drop (m+1) (a:as)) = (a:as)`.

```
  a:(take m as ++ drop (m+1) (a:as)) = (a:as)
```

≡ { axiom of equality of lists (Note 1) }

```
  take m as ++ drop (m+1) (a:as) = as
```

≡ { $m \geq 0$, specification }

```
  take m as ++ drop (m+1) (a:as) = take m as ++ drop m as
```

≡ { equality of lists (Note 2) }

```
  drop (m+1) (a:as) = drop m as
```

Because of the induction hypothesis, we know that `drop m as` is defined. This gives a definition for this case.

Notes:

0. The symbol ≡ denotes logical equivalence (i.e. if and only if) and is pronounced "equivales".
1. `(x:xs) = (y:ys)` ≡ `x = y && xs = ys`. In this case `x` and `y` both equal `a`.
2. `xs ++ ys = xs ++ zs` ≡ `ys = zs` can be proved by induction on `xs` using the Note 1 property.

Bringing the cases together, we get the definition that we saw earlier.

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs             -- drop.1
drop _ []          = []             -- drop.2
drop n (_:xs)      = drop (n-1) xs  -- drop.3
```

26.6 Tail Recursion Theorem

In Chapter 14, we looked at two different definitions of a function to reverse the elements of a list. Function `rev` uses a straightforward backward linear recursive technique and `reverse` uses a tail recursive auxiliary function. We proved that these definitions are equivalent.

```
rev :: [a] -> [a]
rev []       =   []             -- rev.1
rev (x:xs)   =   rev xs ++ [x]  -- rev.2

reverse :: [a] -> [a]
reverse xs   =   rev' xs []                  -- reverse.1
    where rev' [] ys      = ys               -- reverse.2
          rev' (x:xs) ys   = rev' xs (x:ys)  -- reverse.3
```

Function `rev'` is a *generalization* of `rev`. Is there a way to calculate `rev'` from `rev`?

Yes, by using the Tail Recursion Theorem for lists. We develop this theorem in a more general setting than `rev`.

The following is based on Section 4.7 of [Hoogerwoord 1989].

For some types `X` and `Y`, let function `fun` be defined as follows:

```
fun :: X -> Y
fun x | not (b x) = f x               -- fun.1
      | b x       = h x *** fun (g x) -- fun.2
```

- Functions `b`, `f`, `g`, `h`, and `***` are not defined in terms of `fun`.
- `b :: X -> Bool` such that, for any `x`, `b x` is defined whenever `fun x` is defined.
- `g :: X -> X` such that, for any `x`, `g x` is defined whenever `fun x` is defined and `b x` holds.
- `h :: X -> Y` such that, for any `x`, `h x` is defined whenever `fun x` is defined and `b x` holds.
- `(***) :: Y -> Y -> Y` such that operation `***` is defined for all elements of `Y` and is an associative operation with left identity `e`.
- `f :: X -> Y` such that, for any `x`, `f x` is defined whenever `fun x` is defined and `not (b x)` holds.
- `X` with relation $\prec$ admits induction (i.e. $\langle$`X`$, \prec\rangle$ is a *well-founded ordering*).
- For any `x`, if `fun x` is defined and `b x` holds, then `g x` $\prec$ `x`.

Note that both `fun x` and the recursive leg `h x *** fun (g x)` have the general structure `y *** fun z` for some expressions `y` and `z` (i.e. `fun x = e *** fun x`). Thus we specify a more general function `fun'` bsuch that

```
fun' :: Y -> X -> Y
fun' y x = y *** fun x
```

and such that `fun'` s is defined for any `x` $\in$ `X` for which `fun x` is defined.

Given the above specification, we note that:

```
  fun' e x
```

= { `fun'` specification }

```
  e *** fun x
```

= { `e` is the left identity for `***` }

```
  fun x
```

We proceed by induction on the type `X` with $\prec$. (We are using *well-founded induction*, a more general form of induction than we have used before.

We have two cases. The base case is when `not (b x)` holds for argument `x` of `fun'`. The inductive case is when `b x` holds (i.e, `g x` $\prec$`x`).

Base case `not (b x)`: (That is, `x` is a minimal element of `X`under $\prec$.)

```
  fun' y x
```

= { `fun'` specification }

```
  y *** fun x
```

= { `fun.1` }

```
y *** f x
```

Inductive case `b x`: (That is, `g x` $\prec$ `x`.)

Given that there is a definition for `fun' y (g x)` that satisfies the specification for any `y`

```
fun' y (g x) = y *** fun (g x)
```

calculate a definition for `fun' y x` that satisfies the specification.

```
fun' y x
```

= { `fun'` specification }

```
y *** fun x
```

= { `fun.2` }

```
y *** (h x *** fun (g x))
```

= { `***` associativity }

```
(y *** h x) *** fun (g x)
```

= { `g x` $\prec$ `x`, induction hypothesis }

```
fun' (y *** h x) (g x)
```

Thus we have synthesized the following tail recursive definition for function `fun'` and essentially proved the Tail Recursion Theorem shown below.

```
fun' :: Y -> X -> Y
fun' y x | not (b x) = y *** f x              -- fun'.1
         | b x       = fun' (y *** h x) (g x) -- fun'.2
```

Note that the first parameter of `fun'` is an *accumulating parameter*.

Tail Recursion Theorem: If `fun`, `fun'`, and `e` are defined as given above, then `fun x = fun' e x`.

Now let's consider the `rev` and `rev'` functions again. First, let's rewrite the definitions of `rev` in a form similar to the definition of `fun`.

```
rev :: [a] -> [a]
rev xs | xs == [] = []                          -- rev.1
       | xs /= [] = rev (tail xs) ++ [head xs]  -- rev.2
```

For `rev` we substitute the following for the components of the `fun` definition:

- `fun x` $\leftarrow$ `rev xs`
- `b x` $\leftarrow$ `xs /= []`
- `g x` $\leftarrow$ `tail xs`
- `h x` $\leftarrow$ `[head xs]`

- `l *** r` ← `r ++ l` (Note the flipped operands,)
- `f x` ← `[]`
- `l` ≺ `r` ← `(length l) < (length r)`
- `e` ← `[]`
- `fun' y x` ← `rev' xs ys` (Note the flipped arguments.)

Thus, by applying the tail recursion theorem, `fun'` becomes the following:

```
rev' :: [a] -> [a] -> [a]
rev' xs ys
    | xs == [] = ys                                  -- rev'.1
    | xs /= [] = rev' (tail xs) ([head xs]++ys)      -- rev'.2
```

From the Tail Recursion Theorem, we conclude that `rev xs = rev' xs []`.

Why would we want to convert a backward linear recursive function to a tail recursive form?

- A tail recursive definition is sometimes more space efficient (as we saw in Chapter 9).

  This is especially the case if the strictness of an accumulating parameter can be exploited (as we saw in Chapters 9 and 15).

- A tail recursive definition sometimes allows the replacement of an "expensive" operation (requiring many steps) by a less "expensive" one. (For example, `++` is replaced by cons in the transformation from `rev` to `rev'`.)
- A tail recursive definition can be transformed (either by hand or by a compiler) into an efficient loop.
- A tail recursive definition is usually more general than its backward linear recursive counterpart. Sometimes we can exploit this generality to synthesize a more efficient definition. (We see an example of this in the next subsection.)

26.7 Finding Better Tail Recursive Algorithms

This section is adapted from Section 11.3 of Cohen's textbook [Cohen 1990].

Although the Tail Recursion Theorem is important, the technique we used to develop it is perhaps even more important. We can sometimes use the technique to transform one tail recursive definition into another that is more efficient [Hoogerwoord 1989].

Consider exponentiation by a natural number power. The operation `**` can be defined recursively in terms of multiplication as follows:

```
infixr 8 **
(**) :: Int -> Int -> Int
m ** 0         = 1              -- **.1
m ** n | n > 0 = m * (m ** n)   -- **.2
```

For `(**)` we substitute the following for the components of the `fun` definition of the previous subsection:

- `fun x` ← `m ** n`
- `b x` ← `n > 0` (Applied only to natural numbers.)
- `g x` ← `n - 1`
- `h x` ← `m`
- `l *** r` ← `l * r`
- `f x` ← `1`
- `l` ≺ `r` ← `l < r`
- `e` ← `1`
- `fun' y x` ← `exp a m n`

Thus, by applying the Tail Recursion Theorem, we define the function `exp` such that

```
exp a m n = a * (m ** n)
```

and, in particular:

```
exp 1 m n = m ** n
```

The resulting function `exp` is defined as follows (for `n >= 0`):

```
exp :: Int -> Int -> Int -> Int
exp a m 0 = a               -- exp.1
exp a m n = exp (a*m) m n   -- exp.2
```

In terms of time, this function is no more efficient than the original version; both require `O(n)` multiplies. (However, by exploiting the strictness of the first parameter, `exp` can be made more space efficient than `**`.)

Note that `exp` algorithm converges upon the final result in steps of one. Can we take advantage of the generality of `exp` and the arithmetic properties of exponentiation to find an algorithm that converges in larger steps?

Yes, we can by using the technique that we used to develop the Tail Recursion Theorem. In particular, let's try to synthesize an algorithm that converges logarithmically (in steps of half the distance) instead of linearly.

Speaking operationally, we are looking for a "short cut" to the result. To find this short cut, we use the "maps" that we have of the "terrain". That is, we take advantage of the properties we know about the exponentiation operator.

We thus attempt to find expressions `x` and `y` such that

```
exp x y (n/2) = exp a m n
```

where "`/`" represents division on integers.

For the base case where `n = 0`, this is trivial. We proceed with a calculation to discover values for `x` and `y` that make

```
exp x y (n/2) = exp a m n
```

when `n > 0` (i.e. in the inductive case). In doing this we can use the specification for `exp` (i.e.`exp a m n = a * (m ** n)`).

```
exp x y (n/2)
```

= { `exp` specification }

```
x * (y ** (n/2))
```

= { Choose `y = m ** 2` (Note 1) }

```
x * ((m ** 2) ** (n/2))
```

Note 1: The strategy is to make choices for `x` and `y` that make

```
x * (y ** (n/2))
```

and

```
a * (m ** n)
```

equal. This choice for `y` is toward getting the `m ** n` term.

Because we are dealing with integer division, we need to consider two cases because of truncation.

Subcase `even n` (for `n > 0`):

```
x * ((m ** 2) ** (n/2))
```

= { arithmetic properties of exponentiation, `n` even }

```
x * (m ** n)
```

= { Choose `x = a`, toward getting `a * (m ** n)` }

```
a * (m ** n)
```

= { `exp` specification }

```
exp a m n
```

Thus, for even `n`, we derive:

```
exp a m n = exp a (m*m) (n/2)
```

We optimize and replace `m ** 2` by `m * m`.

Subcase `odd n` (for `n > 0`): That is, `n/2 = (n-1)/2`.

```
    x * ((m ** 2) ** ((n-1)/2))
```

= { arithmetic properties of exponentiation }

```
    x * (m ** (n-1))
```

= { Choose `x = a * m`, toward getting `a * (m ** n)` }

```
    (a * m) * (m ** (n-1))
```

= { arithmetic properties of exponentiation }

```
    a * (m ** n)
```

= { `exp` specification }

```
    exp a m n
```

Thus, for odd `n`, we derive:

```
    exp a m n = exp (a*m) (m*m) (n/2)
```

To differentiate the logarithmic definition for exponentiation from the linear one, we rename the former to `exp'`. We have thus defined `exp'` as follows (for `n >= 0`):

```
exp' :: Int -> Int -> Int -> Int
exp' a m 0            = a                  -- exp'.1
exp' a m n
    | even n = exp' a     (m*m) (n/2)      -- exp'.2
    | odd n  = exp' (a*m) (m*m) ((n-1)/2)  -- exp'.3
```

Above we showed that `exp a m n = exp' a m n`. However, execution of `exp'` converges faster upon the result: `O(log2(n))` steps rather than `O(n)`.

Note: Multiplication and division of integers by natural number powers of 2, particularly 2^1, can be implemented on must current computers by arithmetic left and right shifts, respectively, which are faster than general multiplication and division.

26.8 What Next?

The next chapter applies the program synthesis techniques developed in this chapter to a larger set of examples on text processing.

No subsequent chapter depends explicitly upon the program synthesis content from these chapters. However, if practiced regularly, the techniques explored in this chapter can enhance a programmer's ability to solve problems and construct correct functional programming solutions.

26.9 Exercises

1. The following function computes the integer base 2 logarithm of a positive integer:

```
lg :: Int -> Int
lg x | x == 1 = 0
     | x > 1  = 1 + lg (x/2)
```

   Using the tail recursion theorem, write a definition for `lg` that is tail recursive.

2. Synthesize the recursive definition for `++` from the following specification:

```
xs ++ ys  =  foldr (:) ys xs
```

3. Using tupling and function `fact5` from Chapter 4, synthesize an efficient function `allfacts` to generate a list of factorials for natural numbers 0 through parameter `n`, inclusive.

4. Consider the following recursive definition for natural number multiplication:

```
mul :: Int -> Int -> Int
mul m 0     = 0
mul m (n+1) = m + mul m n
```

   This is an `O(n)` algorithm for computing `m * n`. Synthesize an alternative operation that is `O(log2(n))`. Doubling (i.e. `n*2`) and halving (i.e `n/2` with truncation) operations may be used but not multiplication (`*`) in general.

5. Derive a "more general" version of the Tail Recursion Theorem for functions of the shape

```
func ::  X -> Y
func x | not (b x) = f x              -          -- func.1
       | b x       = h x *** func (g x) +++ d.x -- func.2
```

   where functions `b`, `f`, `g`, and `h` are constrained as in the definition of `fun` in the Tail Recursion Theorem. Be sure to identify the appropriate constraints on `d`, `***`, and `+++` including the necessary properties of `***` and `+++`.

26.10 Acknowledgements

In Summer 2018, I adapted and revised this chapter and the next from:

- chapter 12 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]

These previous notes drew on the presentations in the first edition of the Bird and Wadler textbook [Bird 1988], Hoogerwoord's dissertation [Hoogerwoord 1989], Kaldewaij's textbook [Kaldewaij 1990], Cohen's textbook [Cohen 1990], and other sources.

I incorporated this work as new Chapter 26, Program Synthesis (this chapter), and new Chapter 27, Text Processing, in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

26.11 References

[Bird 1988]: Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[Bird 1998]: Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[Bird 2015]: Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

[Cohen 1990]: Edward Cohen. *Programming in the 1990's: An Introduction to the Calculation of Programs*, Springer-Verlag, 1990.

[Cunningham 2014]: H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[Gries 1981]: David Gries. *Science of Programming*, Springer 1981.

[Hoogerwoord 1989]: Rob Hoogerwoord. *The Design of Functional Programs: A Calculational Approach*, PhD Dissertation, Eindhoven Technical University, Eindhoven, The Netherlands, 1989.

[Kaldewaij 1990]: Anne Kaldewaij. *Programming: The Derivation of Algorithms*, Prentice Hall International, 1990.

26.12 Terms and Concepts

TODO