Exploring Languages with Interpreters and Functional Programming Chapter 29

H. Conrad Cunningham

6 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 document requires use of a browser that supports the display of MathML. A good choice as of August 2018 is a recent version of Firefox from Mozilla.

29 Divide and Conquer Algorithms

29.1 Chapter Introduction

This Chapter consists of a translation of partial chapter 14 of [Cunningham 2014]. It is incomplete.

TODO: Intro

Reference: This chapter is based on section 6.4 of the Bird and Wadler textbook [Bird 1988] and section 4.2 of Kelly's dissertation [Kelly 1989].

29.2 Overview

The general strategy strategy for *divide-and-conquer algorithms* is as follows:

1. *Decompose* problem P into subproblems, each like P but with a smaller input argument.
2. *Solve* each subproblem, either directly or by recursively applying the strategy.
3. *Assemble* the solution to P by combining the solutions to its subproblems.

The advantages of divide-and-conquer algorithms are that they:

- can lead to efficient solutions.
- allow use of a "horizontal" parallelism. Similar problems can be

Section 6.4 of the Bird and Wadler textbook [Bird 1988] discusses several important divide and conquer algorithms: mergesort, quicksort, multiplication, and binary search.

In these algorithms the divide and conquer technique leads to more efficient algorithms.

For example, a simple *sequential search* is `O(n)` (where `n` denotes the length of the input). Application of the divide-and-conquer strategy leads to the *binary search* which is a more efficient, a `O(log2(n))` search.

As a general pattern of computation, the divide and conquer strategy can be stated with the following higher order function:

```haskell
divideAndConquer :: (a -> Bool)           -- trivial
                    -> (a -> b)           -- simplySolve
                    -> (a -> [a])         -- decompose
                    -> (a -> [b] -> b)    -- combineSolutions
                    -> a                  -- problem
                    -> b
```

```
divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
    = solve problem
      where solve p
          | trivial p = simplySolve p
          | otherwise = combineSolutions p
                            (map solve (decompose p))
```

If the problem is trivially simple (i.e. `trivial p`), then it is solved directly using `simplySolve`.

If the problem is not trivially simple, then it is decomposed into a list of subproblems using `decompose` and each subproblem is solved separately using `map solve`. The list of solutions to the subproblems are then assembled into a solution for the problem using `combineSolutions`.

Sometimes `combineSolutions` may require the original problem description in order to put the solutions back together properly. Hence the parameter `p`.

Note that the solution of each subproblem is completely independent from the solution of all the others.

If all the subproblem solutions are needed by `combineSolutions`, then the language implementation could potentially solve the subproblems simultaneously. The implementation could take advantage of the availability of multiple processors and actually evaluate the expressions in parallel. This is "horizontal" parallelism as described above.

If `combineSolutions` does not require all the subproblem solutions, then the subproblems cannot be safely solved in parallel. If they were, the result of `combineSolutions` might be *nondeterministic*, that is, the result could be dependent upon the relative order in which the subproblem results are completed.

29.3 Divide and Conquer Fibonacci

Now let's use the function `divideAndConquer` to define a few functions.

First, let's define a Fibonacci function. (This is adapted from the function defined on pages 77-8 of Kelly [Kelly 1989]. This function is inefficient. It is given here to illustrate the technique.)

```
fib :: Int -> Int
fib n = divideAndConquer trivial simplySolve decompose
                         combineSolutions problem
        where trivial 0                = True
              trivial 1                = True
              trivial (m+2)            = False
              simplySolve 0            = 0
              simplySolve 1            = 1
```

```
decompose m              = [m-1,m-2]
combineSolutions _ [x,y] = x + y
```

29.4 Divide and Conquer Folding

Next, let's consider a folding function (similar to `foldr` and `foldl`) that uses the function `divideAndConquer`. (This is adapted from the function defined on pages 79-80 of Kelly [Kelly 1989].)

```
fold :: (a -> a -> a) -> a -> [a] -> a
fold op i =
    divideAndConquer trivial simplySolve decompose
                     combineSolutions
    where trivial xs               = length xs <= 1
          simplySolve []          = i
          simplySolve [x]         = x
          decompose xs            = [take m xs, drop m xs]
                                    where m = length xs / 2
          combineSolutions _ [x,y] = op x y
```

This function divides the input list into two almost equal parts, folds each part separately, and then applies the operation to the two partial results to get the combined result.

The `fold` function depends upon the operation `op` being *associative*. That is, the result must not be affected by the order in which the operation is applied to adjacent elements of the input list.

In `foldr` and `foldl`, the operations are not required to be associative. Thus the result might depend upon the right-to-left operation order in `foldr` or left-to-right order in `foldl`.

Function `fold` is thus a bit less general. However, since the operation is associative and `combineSolutions` is strict in all elements of its second argument, the operations on pairs of elements from the list can be safely done in parallel,

Another divide-and-conquer definition of a folding function follows. Function `fold'` is an optimized version of `fold`.

```
fold' :: (a -> a -> a) -> a -> [a] -> a
fold' op i xs = foldt (length xs) xs
                where foldt _ []  =   i
                      foldt _ [x] =   x
                      foldt n ys  = op (foldt m (take m ys))
                                       (foldt m' (drop m ys))
                                    where m  = n / 2
                                          m' = n - m
```

29.5 Minimum and Maximum of a List

Consider the problem of finding both the minimum and the maximum values in a nonempty list and returning them as a pair.

First let's look at a definition that uses the left-folding operator.

```
sMinMax :: Ord a => [a] -> (a,a)
sMinMax (x:xs) = foldl' newmm (x,x) xs
                 where newmm (y,z) u = (min y u, max z u)
```

Let's assume that the comparisons of the elements are expensive and base our time measure on the number of comparisons. Let `n` denote the length of the list argument and `time` be a time function

A singleton list requires no comparisons. Each additional element adds two comparisons (one `min` and one `max`).

```
time n | n == 1 = 0
       | n >= 2 = time (n-1) + 2
```

Thus `time n == 2 * n - 2`.

Now let's look at a divide and conquer solution.

```
minMax :: Ord a => [a] -> (a,a)
minMax [x]   = (x,x)
minMax [x,y] = if x < y then (x,y) else (y,x)
minMax xs    = (min a c, max b d)
               where m     = length xs / 2
                     (a,b) = minMax (take m xs)
                     (c,d) = minMax (drop m xs)
```

Again let's count the number of comparisons for a list of length `n`.

```
time n | n == 1 = 0
       | n == 2 = 1
       | n > 2  = time (floor (n/2)) + time (ceiling (n/2)) + 2
```

For convenience suppose `n = 2^k` for some `k >= 1`.

```
time n = 2 * time (n/2) + 2
       = 2 * (2 * time (n/4) + 2) + 2
       = 4 * time (n/4) + 4 + 2
       = ...
       = 2^(k-1) * time 2 + sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2 * sum [ 2^i | i <- [1..(k-1)] ]
                 -     sum [ 2^i | i <- [1..(k-1)] ]
       = 2^(k-1) + 2^k - 2
       = 3 * 2^(k-1) - 2
       = 3 * (n/2) - 2
```

Thus the divide and conquer version takes 25 percent fewer comparisons than the left-folding version.

So, if element comparisons are the expensive in relation to to the `take`, `drop`, and `length` list operations, then the divide and conquer version is better. However, if that is not the case, then the left-folding version is probably better.

Of course, we can also express `minMax` in terms of the function `divideAndConquer`.

```
minMax' :: Ord a =$> [a] -> (a,a)
minMax' = divideAndConquer trivial simplySolve decompose
                           combineSolutions
          where n                 = length xs
                m                 = n/2
                trivial xs        = n <= 2
                simplySolve [x]   = (x,x)
                simplySolve [x,y] =
                    if x < y then (x,y) else (y,x)
                decompose xs =
                    [take m xs, drop m xs]
                combineSolutions _ [(a,b),(c,d)] =
                    (min a c, max b d)
```

29.6 What Next?

TODO

29.7 Exercises

TODO

29.8 Acknowledgements

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

- chapter 14 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]

These previous notes drew on the presentations in the 1st edition of the Bird and Wadler textbook [Bird 1988], Kelly's dissertation [Kelly 1989], and other sources.

I incorporated this work as new Chapter 29, Divide and Conquer Algorithms, 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.

29.9 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.

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Kelly 1989**]**:** Paul H. J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors*, MIT Press, June 1989.

[**Thompson 1996**]**:** Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

[**Wentworth 1990**]**:** E. P. Wentworth. *Introduction to Functional Programming using RUFL*, Department of Computer Science, Rhodes University, Grahamstown, South Africa, August 1990.

29.10 Terms and Concepts

Divide and conquer, horizontal parallelism, divide and conquer as higher-order function, sequential search binary search, simply solve, decompose, combine solutions, Fibonacci sequence, nondeterministic, associative.