Noname manuscript No.
(will be inserted by the editor)

Getting Lazy and Pure in Code Contests by Using Haskell

Chen Huo

Received: date / Accepted: date (Regular Research Paper)

Abstract Lazy purely functional languages, like Haskell, are never the first choices for code contests or competitive programming. We studied 107 problems from online code contest platforms, and found that Haskell users do not yet have standardized solutions for common situations in code contests due to the limitations of being lazy and pure. To name some, with side-effects prohibited (pure), it is tricky to do IO and write graph algorithms under time complexity requirements. To help laziness and purity reconcile with code contests, we derive an innovative collection of template solutions inspired by both the functional programming literature and actual user solutions from online code contest platforms. The collection will serve as an entry point for functional programming learners to code contests and a showcase of Haskell usage in this domain.

Keywords Computer Science Education · Code Contests · Programming Languages · Functional Programming

1 Introduction

It is challenging to use a lazy purely functional language like Haskell in code contests like International Collegiate Programming Contest (ICPC) (if supported). During a time-limiting contest, contestants are only allowed to browse standard language documentations online and to bring hard-copy materials with them. At the same time, for a programming language, being lazy brings uncertainty on time complexity and being purely functional means traditional algorithms (i.e., based on mutation) need to be completely rethought[7].

The research community have been working on innovative alternatives of traditional data structures and algorithms for lazy purely functional languages, e.g., [20, 11, 17, 21]. Although many have been implemented and available as packages on Hackage, the Haskell package repository, some key packages are missing in a typical Haskell installation[5] or on online contest platforms like Hackerrank[4]. Typically contests will only provide the "standard" installation for any language. So it is reasonable to assume that, for example, Erwig's inductive graph library[12] will not be an option in code contests.

To make things even harder, most of functional programming textbooks focus on explaining the fundamental programming and language concepts[12]. As one can imagine, potential Haskell contestants are struggling[8, 9]. Complaints range from lacking templates for IO to lacking feasible graph algorithms to follow. Meanwhile, the imperative counterparts, like C or Java, can easily translate algorithms in pseudo code to working code in contests. Template solutions can be printed

C. Huo
Software Engineering
Shippensburg University of Pennsylvania
MCT150, 1871 Old Main Dr., Shippensburg PA, 17257 USA
Tel.: +1-717-4771642
E-mail: chuo@engr.ship.edu

and brought to the contests. Given such adversities, fewer contestants will use Haskell, and thus fewer contests will support Haskell, entering a death spiral. To increase *diversity*, the goal of the study is to narrow down the gap between Haskell and its imperative counterparts in code contests: Can we derive a collective solution template from only standard Haskell packages, within the limits of offline code contests?

To answer the question, we studied 107 problems from online code contest platforms. For each problem, we solve it first and then compare the solution with other Haskell submissions. We confirm that Haskell has no uniform solutions like their imperative counterparts (see Section 3.1 as an example). From the collection of community submissions and research literature, we distilled solution templates in three major categories: IO actions, sample functional data structure usages, and graph algorithms, conforming to the limits in code contests.

The rest of the paper is arranged as follows: Section 2 gives brief background on code contests of our interest, lazy purely functional languages, and how the proposed template solutions are chosen. More specific related work will be given in each of the 3 following sections. Section 3 discusses inputs and outputs for typical code contest problems. The proposed IO template uses the `do` notation so that they resemble their imperative counterparts. Section 4 briefly introduces functional data structures with two case studies: dynamic programming and disjoint sets. Section 5 first studies functional graph representations and algorithms. We present a hybrid of King's[16] and Erwig's[12] methods. Lastly, Section 6 summarizes the study and the outlook of future work. The appendices contain the template for contestants to bring to contests and a full list of the problems studied from online platforms[1].

2 Background

2.1 Code Contests

We prepare the templates in this study for offline contests similar to ICPC. For each problem to solve, contestants will read in text inputs from standard input of certain layout and print out the answers to standard output in certain format. The problems can be described as *algorithmic* problems (e.g., as opposed to *program design*). For example, a problem may describe a map of cities connected by highways with tolls and ask what's the route with the least toll cost from one city to another. Contestants need to find out that it's the Dijsktra Shortest Path in disguise. In this environment, this is crucial to have common IO templates for contest-like inputs and outputs and have templates for common algorithms like Dijsktra. Since there's no limit on printed documents, it's common practice that teams will bring printed template code of their choice to the contests.

There are a number of online code contest platforms which offer similar problems and test cases. We surveyed them and found that Hackerrank[4] and Codewars[1] are the two with functioning Haskell support. Hackerrank covers a wide range of problem domains and has the largest user base. It provides most of problems used in this study (100). Although problems from Codewars do not resemble code contest problems well, we included 7 problems on it to increase diversity. One of the reasons that we choose problems from online platforms is their availability. Readers will need the original problems to reinforce their understandings of our templates. Unlike the ICPC problems which may not be available to everyone, readers can find the problems referenced in this paper from Hackerrank by their names.

2.2 Lazy Purely Functional Languages

A language is purely-functional when side-effects are never allowed. In this sense, O'Caml is almost pure since it allows reference types. Haskell is completely pure and IO actions are simulated by monads[25]. It is not a surprise that the common array types in the C family do not fit well in Haskell. That means we must rework on many algorithms from traditional algorithm textbooks like [10]. (Some Haskell data structures do support true in-place mutation, e.g., `STArray`, which

[1] Due to limit of space, appendices can be accessed on `https://web.engr.ship.edu/~chuo/appendices.pdf`.

give users a chance to directly translate from C-like code. However, in this study, we'd like to see how purity can be retained in code contests.)

Lazy stands for lazy evaluation or *call-by-need*. In lazy evaluation, an expression is not evaluated until it is needed. Once it's evaluated, its result is memoized. There are a number of publications on the benefits of laziness, e.g., [14, 25, 15]. In the following sections, we will demonstrate, case by case, how laziness help write compact code. At the same time, laziness leaves not much control to programmers for when and whether calculations will take place. In an imperative language like C, the time complexity of statement `f(g(x))` can be calculated simply as the sum of the time cost of `g(x)` and `f(y)`, where `y` is the result of `g(x)`. It is clearly not wise to find the minimum element in a list by taking the first element after sorting the list. That would take $O(N^2)$ time using insertion sort. However, for a lazy language, the same approach may only take $O(N)$ time[24]. Under lazy evaluation, insertion sort behaves like selection sort, so we can take the minimum as soon as it moves to the front of the list[7].

It appears that, to use Haskell in code contests, one shall be familiar with monadic IO, find alternatives to traditional algorithms based on array mutations, and have fundamental understandings on time complexity in a lazy language. This study proposes to standardize the solutions for these burden so that the contestants can focus on problem solving like other imperative language users.

2.3 Evaluating The Solutions

Possible Haskell code templates are assessed in three facets: 1) Does it fit for code contests? That is, easy to reference and type from hard-copies, and cover a range of specific problem domains in code contests. 2) Does it have style? That is, being lazy, purely-functional, and modularized. 3) Is it efficient? That is, do solutions from template code give reasonable asymptotic time complexity and satisfy time limits of test cases from online platforms? To our best knowledge, this study is the first that provides collective solutions regarding the above three characteristics.

3 Input and Output

3.1 Background

Inputs and outputs are typical side-effects. As a purely functional language, Haskell does IO through Monads[22]. The consequence is that unlike the imperative counterparts which print out "Hello World" on the first page, Haskell textbooks[15, 18, 23] typically bring it up near halfway through the contents, by the do-notation. Readers will learn even later that do-notations are syntactic sugar for monadic binds.

The process is highly volatile and not standardized across different users and problems from Hackerrank submissions. It is too demanding to cope with function composition `(.)` and monadic binding `(>>=)` just for code contests. Fig. 1 shows the first four Haskell submissions for *Longest Common Child* problem on Hackerrank. To take a closer look at the first main in the listing: `lcs` is the function that solves the problem. `(<$>)` and `(<*>)` are applicative functors to "lift" `lcs` in the IO context, just to extract inputs from the two getlines. The first `(=<<)` is just the flipped `(>>=)`, the monadic bind.

```
main = print =<< lcs <$> getLine <*> getLine
main = print =<< solve. B.lines =<< B.getContents
main = print . solve . take 2 . lines =<< getContents
main = B.interact $
       B.pack <$> show <$> solveBase5 <$> (B.unpack <$>) <$>
       B.lines
```

Fig. 1: Submissions by *BlueD, trimerous, wood6, rhovland* are all different.

All this effort is to get two strings from two lines, and output a single result string. For an IO action as simple as this, there is not much consistency in Haskell submissions. On the other hand, one can find that almost every C submission uses `scanf("%s %s", s1, s2)` and `printf`. That's it. Haskell users are dealing with so much adversity even before starting solving the actual problem. This clearly calls for an intuitive template to follow.

Crux: Can we have a template for typical code contest inputs and outputs without involving too many `>>=` and `.`?

3.2 Input Template

A fifth submission by *akegalj* coincides our goal except we also illustrate the use of `let` in the code snippet below.

```
main = do
  s1 <- getLine
  s2 <- getLine
  let ans = lcs s1 s2
  print ans
```

Oftentimes you will be asked to read a list of numbers from a line. We present `readInts` as a reusable component in Fig. 2. Although there are function compositions and nested fmaps, a contestant can use it as a black box. Another important component we identified, among tens of monadic functions, is `replicateM`. This function repeats a monadic computation for a given number of times and returns a list in some monad. Equipped with our `readInts` and `replicateM`, we can use them in various combinations for a large number of code contest problems. Given the example input below, we can use the template IO code in Fig. 2.

```
5 2             % m numbers in a line, n lines
1 2 3 4 5
2 3 4 5 6
```

```
import Control.Monad (replicateM)

readInts :: IO [Int]
readInts = fmap (fmap read . words) getLine

main = do
  [_, n] <- readInts              -- n lines, don't care about m
  nss    <- replicateM n readInts -- nss is a list of lists
  let ns1 = head nss1              -- first list
  print $ size ns1                 -- print out the size
```

Fig. 2: Example use of `readInts` and `replicateM`. The `do` block can be nested.

Besides the two components mentioned above, the built-in `read` is a versatile tool. Its type `Read a => String -> a` tells you that it can interpret a string as whichever type you want, as long as it has a `Read` instance. In our `readInts`, `read` is used on a numeric string, e.g., `"3"`, separated from a line by `words`. The resulting type, i.e., `IO [Int]`, instructs `read` to interpret the string as an integer. A variant is `readLn :: Read a => IO a` which reads a line directly from input, e.g., `d <- readLn :: IO Double`. Curious readers may wonder if `readLn :: IO [Int]` can replace our `readInts`. It turns out the former only recognizes strings like `"[1,2,3]"`, the default string form for lists.

3.3 Large Inputs

Oftentimes a problem will tell you the maximum size of inputs. When the input is expected to be large, the performance of String IO may suffer due to the fact that the `String` type is a linked list of characters. `ByteString` (as `B`) provides low-level IO functions with identical names. We can first read in byte strings and then use `ByteString.Char8` (as `BC`) utilities to convert the byte string to wanted types. `readInts` can be written as follows without modifying other parts of the program.

```
readInts' :: IO [Int]
readInts' = B.unfoldr (BC.readInt . BC.dropWhile (== ' ')) <$> B.getLine
```

As an example, one test case of *Dijkstra: Shortest Reach 2* has over 3 million edges for the input graph and the input text file is about 36 *MB* in size. While using the regular String IO, it takes roughly 15 seconds to finish reading the input. However, it only takes 1.5 seconds with the byte-string version. We see one order of magnitude improvement by just swithcing to `readInt'`.

3.4 Outputs

The output side is generally less complex. Readers need to use `putStr` for string data type and `print` for the other types, as `print` will also print out the quotes for strings. A common scene is to print out lists or list of lists, separated by spaces or lines. The process is the dual to `readInts`. We convert each item to a string by `show` and then assemble them using `unwords` or `unlines`. For example:

```
-- print a list [1,2,3] to 1 2 3
putStr $ (unwords . fmap show) ns
-- print list of lists in separate lines
putStr $ unlines (fmap (unwords . fmap show) nss)
```

Readers might be surprised to see the old good `printf`. It is similar to `read` that `printf` "prints" to whichever type you like as long as there's an instance for `PrintfType`. For example, we can:

```
printf "%-3d %.6f" 3 3.14
```

3.5 Conclusion

Contestants can use the customized `readInts` and other components or variants to standardize the IO part by just the do-notation. The use of `let` expressions can help process the inputs without getting out of the monad.

4 Data Types

4.1 Related Work

As a purely function language, Haskell often has immutable tree-like alternatives to the imperative counterparts. For example, arrays, sets, and heaps can be implemented by tree-like structures, e.g., [21, 20]. Time complexity analysis with lazy evaluation is challenging. Luckily, Okasaki[20] summarizes techniques for analyzing and constructing lazy functional data structures. For the Haskell collection types, many operations can have similar amortized costs to their imperative counterparts, while some may not. Contestants shall choose the right data structure with care.

Crux: How does one use the lazy and pure Haskell collection alternatives in code contests? In this section, we present two use cases of Haskell array and set operations, *dynamic programming* and *Kruskal (minimal spanning tree)*.

4.2 Array: Dynamic Programming

For Haskell arrays, elements can accessed by `(!)`, e.g., `arr ! i`, which takes constant time (when the indices are integers or alike). Updates are more efficient if done in a batch by `(//)`. When `(//)` takes in an array of size N and a list of index-value pairs of size M, the time cost is $O(N + M)$.

Dynamic programming is a technique that memoizes results of sub-problems. The structure for storing the results is usually an array. The issue with immutable arrays is that once we initialize the array, it will be inefficient to change array elements later in the process (for M will not be large each time). Luckily we can leverage the lazy constructor `array`.

```
array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

The first parameter is a 2-tuple of the index range. We have the flexibility to make the array 0-indexed or 1-indexed. The later can be convenient for a graph presentation since the node numbers often starts at 1 in code contests. The second parameter is a list of index-value pairs to be populated to the array. Laziness allows us to access array elements on the fly during the construction as long as the accessed elements are ready. Fig. 3 shows the classic Fibonacci sequence problem. In Fig. 3b, the local function `go` has the shape of the recursive `fib` in Fig. 3a except that recursive calls are replaced with array subscripts. The array `arr` calls `go` in the middle of construction.

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

(a) Recursive version of `fib`

```
fib n = arr ! n
  where go 0 = 1
        go 1 = 1
        go n = arr ! (n - 1) + arr ! (n - 2)
        arr  = array (0, n)
               [(i, go i) | i <- [0..n]]
```

(b) Dynamic programming version of `fib`. `go` has the shape of recursive `fib`. Recursive function calls are replaced with array subscripts.

Fig. 3: Replacing the function calls with array subscripts.

This technique coincides with Tikhon's blog on dynamic programming[6]. The blog post summarizes a mechanical process as follows: (1) add an array at the same scope level as the recursive function, (2) define each array element as a call back into the function with the appropriate index, and (3) replace each recursive call with an index into the array. We show a 2-dimensional example, the *Coin Change* problem on Hackerrank in Fig. 4: How many ways are there to change `m` cents with a list of (sorted) coin denominations `denoms`? Line 7 to 10 resembles the recursive solution with function calls replaced with array subscripts. Line 6 constructs the array with a variant `listArray` which omits the index part in the list elements by assuming the default order.

We sampled 20 submissions among about 100 passing submissions on Hackerrank. About a third of them used this technique. Other methods include using a `List`, using a `Map`, and folding a `List`. List indexing is $O(N)$ and $O(log(N))$ for `Map`. The time limit on test cases cannot exclude these solutions. The `List` folding method may have better space efficiency but is harder to follow and generalize. It is similar to the method used in [23] who then pointed us to `Data.Array`.

4.3 Set: Kruskal Minimal Spanning Tree

The *Kruskal* algorithm can compute a minimal spanning tree (MST) with the set of graph edges. The algorithm needs the edges sorted by their weights, a *disjoint (union-find) set*. Here we present a quick implementation of disjoint sets using a list of `IntSet` in Fig. 5, a program that computes the weight of an MST in a graph. We use the built-in Set `union` function to union two sets. For *find*, list `partition` is used on line 11 for finding sets containing nodes u or v. The resulting `uv` must either be empty or contain two sets, as shown on line 7 and 8. Membership and insertion for an `IntSet` take $O(min(n, W))$ time where W is number of bits in `Int` (32 or 64). Union and list

```haskell
change :: [Int] -> Int -> Integer
change denoms m = c ! (n, m)
  where bounds = ((0,0),(length denoms, m))
        ds     = listArray (1,n) denoms
        n      = length denoms
        arr    = listArray bounds [go i j | (i,j) <- range bounds]
        go i j | i == 0         = 0
               | j == 0         = 1
               | j - (ds!i) < 0 = arr ! (i-1,j)
               | otherwise      = arr ! (i-1,j) + arr ! (i,j-(ds!i))
```

Fig. 4: *Coin Change*: a 2-dimensional dynamic programming example.

partition both take linear time. This implementation is efficient enough in practice that it finishes within 0.15 sec for a graph with 1000 nodes and 10000 edges. In addition, a `Set` (`pq`) is used as a priority queue to obtain the minimums on the fly. Although the Set minimum operation will take $O(logN)$ time, since in this case the minimum is always removed whenever it is queried, on line 10, it is still efficient. There are 9 passing Haskell submission. There are all kinds of implementations using various data types. We found that ours with `IntSet` is more readable and succinct.

```haskell
type DSet = [IntSet]
type Edge = (Int, (Int, Int))

kruskal :: Set Edge -> DSet -> Int -> Int
kruskal pq ds sum
  | Set.null pq    = sum
  | []       <- uv = kruskal pq' (fromList [u,v] : ys) (sum+w)
  | [s1, s2] <- uv = kruskal pq' (union s1 s2 : ys) (sum+w)
  | otherwise      = kruskal pq' ds sum
  where ((w,(u,v)), pq') = Set.deleteFindMin pq
        (uv, ys)         = partition ((||) <$> member u <*> member v) ds
```

Fig. 5: Quick implementations of a priority queue and a disjoint set by `Set` and `IntSet`.

4.4 Conclusion

Laziness gives unusual control flow so that an immutable array can appeared to be mutable during construction. Updates are more efficient than expected when it's done in a batch. `Set` is a typical tree structure so that many operations take logarithm time. `IntSet` operations are close to constant time. Our template in Fig. 5 demonstrated their potentials.

In addition to `Array` and `Set`, the standard installation also includes `Vector` and `Map`. `Vector` is integer-indexed array enriched with plenty of utility functions. `Map` is also a tree-like structure which has $O(logN)$ for common operations. Okasaki[20] includes compact source code for more kinds of lazy functional data structures.

5 Graph Algorithms

5.1 Related Work

Traditional graph representations and algorithms reply heavily on mutations[10]. It is hard for Haskell users to find proper algorithms to follow[8, 9]. Some resort to direct translations from imperative algorithms[3, 19]. From the research community, King explored laziness in breadth and depth first searches in Haskell[17, 16]. Erwig proposed the inductive graph library which enables the more revolutionary "active pattern matching"[13, 11, 12]. Both approaches generate intermediate

data structures from depth or breadth first searches (*dfs* or *bfs*), which subsequent algorithms can reuse. Thanks to laziness, Haskell can use *program fusion* to eliminate much of the overhead for using intermediate representations.

There are several limits in the potential solutions above for code contests. Using mutable data structures like `STArray` in Haskell jeopardizes readability and extendability . King's and Erwig's methods are lazy and purely functional, however, neither is expected to be included in code contests [5]. (The `Data.Graph` module implements King's depth first search algorithms and is available on [4].) Erwig's representation with active pattern matching is too complex to be copied down during code contests, and King's breadth first search family does not have good intermediate representation — Dijisktra is done by mutable arrays[16].

Crux: Can we find lazy and pure graph representations and algorithms that can be feasible to use in code contests? In the rest of the section, we present template algorithms based on depth and breadth first searches. The approach is a hybrid of King's graph representation, depth first search algorithms, and Erwig's breadth first search algorithms.

5.2 Depth First Search

The `Array` graph representation is show on line 1 and 2 in Fig. 6. The intermediate representation of the *dfs* search is `Forest Vertex` which is a list of `Tree`s. A `Tree` is a node and a possibly empty sub-forest. A `Tree` structure is generated by the search on line 10. Interestingly, since `generate` does not keep a record of visited vertices and gets neighbors just by `g!v`, the process is infinite. That's definitely wrong in a non-lazy language, however, it's too soon to say that in Haskell. The infinite forest is trimmed at later stage by `chop`. From the trimmed forest, it takes just a few more lines of code from the forest to topology sort `topSort`, reachability `reachable`, strong/bidirectional connected components `scc/bcc`. Due to limit of space, interested readers can find documentations and source code on [2]. Contestants can trim the source code to code contest needs and keep it as a template for *dfs* algorithms.

```
1  type Vertex = Int
2  type Graph  = Array Vertex [Vertex]
3
4  data Tree a   = Node { rootLabel :: a, subForest :: [Tree a] }
5  type Forest a = [Tree a]
6
7  dfs          :: Graph -> [Vertex] -> Forest Vertex
8  dfs g vs      = prune (bounds g) (map (generate g) vs)
9
10 generate g v  = Node v (map (generate g) (g!v))
```

Fig. 6: A excerpt of King's[16] *dfs* algorithm in `Data.Graph` module.

5.3 Breadth First Search

Erwig[12] proposed a functional intermediate representation, a *root tree* or *rtree*, for breadth first search algorithms. A root tree consists of *root paths*. A root path is just a list from the destination to the source. For a labeled (weighted) graph, a node in a root tree carries the total cost of the path. Fig. 7 shows the Dijisktra Shortest Path algorithm (with a priority heap) in [12] except that we keep track of *unexpanded* nodes by using an `IntSet`. On line 15, a node is removed from the set when it's expanded. On line 10, the algorithm stops when either every node has been expanded or the heap is empty. The latter can happen when some nodes are not reachable from the source. The heap stores root paths with the key being the accumulated weight on the first node. Line 14 "pops" the minimum path and `h'` is the new heap.

Similarly, we use a heap implementation less than 50 lines from [20]. Such functional heaps don't have efficient `decreaseKey`. But we can work around it: the `expand` function updates the costs on the successors and insert them in the heap without being compared with their previous costs. If a duplicate node is in a path with more cost, by the time when this path is popped, this node must have been removed from `vs`, see on line 11. This framework can be used in other breadth search based algorithms such as *Prim*, the *Snakes and Ladders* on Hackerrank, etc.

```
1  type LNode    = (Node, Weight)
2  newtype LPath = LPath [LNode] deriving Show
3  type LRTree   = [LPath]
4
5  expand :: Int -> LPath -> [(Node, Weight)] -> [Heap LPath]
6  expand d (LPath p) = fmap (\(w, d') -> singleton (LPath ((w, d+d'):p)))
7
8  dijkstra :: IntSet -> Heap LPath -> Graph -> LRTree
9  dijkstra vs h g
10   | isEmpty h || S.null vs = []
11   | S.notMember v vs       = dijkstra vs h' g
12   | otherwise              = p:dijkstra vs' h'' g
13   where
14     (p@(LPath ((v,w):_)), h') = findDeleteMin h
15     vs'                       = delete v vs
16     hs                        = expand w p (g ! v)
17     h''                       = mergeAll (h':hs)
```

Fig. 7: Erwig's[12] *bfs* variant for Dijkstra's Shortest Path using array graph representation.

We survey Haskell submissions of *Dijkstra: Shortest Reach 2* on the choices for data structures for 1) visited/unvisited nodes, 2) finding the current minimum path, 3) current shortest path to every node, 4) graph representation. Among the 9 passing Haskell submissions, we can see `STArray`, `Array`, `Map`, `Set` being used for any of the four purposes. This again shows how ad-hoc and volatile the Haskell solutions are in code contests. Our solution is more modularized and more compact, derived from Erwig's work.

5.4 Conclusion

We have a hybrid and tailored solution template for graph representations and algorithms in code contest environments. The template is relatively straightforward to understand, leverages advantages of lazy functional programming, and contains minimum code to write down. Its efficiency has been validated by the online contest platforms.

6 Conclusion and Future Work

Code contests represent a special domain of programming. There are mature solutions using imperative languages and traditional algorithms for this domain. The outcome of the study, the collective template (and why they are chosen), distills the essence of the community solutions and the research literature for a lazy purely functional language, Haskell, to be used in code contests. The template solutions cover a wide range of use cases, retain and leverage laziness and purity, and remain compact to type in contests. The template solutions demonstrated their efficiency by exhibiting reasonable asymptotic time complexity and passing test cases from online platforms under time limits. As there is no "one size fits all" template, readers can expand the template as they want. This study will clear the obstacles for Haskell learners to take a further step by participating in code contests, and can spark more discussions in using lazy purely functional languages in the domain of code contests.

For future work, we can widen the domain of the problems of the study, such as including ICPC problems. We can also explore the possibility of actually having advantages for using a lazy purely functional language in code contests. Finally, if lazy purely functional languages become promising because of this and similar studies, we would like to form directives how to set up the Haskell environment, and advocate more support of functional languages in code contests.

References

1. Code wars. URL `https://www.codewars.com/`
2. Data.Graph. URL `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Graph.html`
3. Dijkstra's algorithm. URL `https://rosettacode.org/wiki/Dijkstra%27s_algorithm`
4. Environment and samples. URL `https://www.hackerrank.com/environment`
5. Included packages. URL `https://www.haskell.org/platform/contents.html`
6. Tikhon Jelvis. URL `https://jelv.is/blog/Lazy-Dynamic-Programming/`
7. Functional algorithm design. Science of Computer Programming **26**(1), 15–31 (1996)
8. Does anyone use haskell (2013). URL `https://codeforces.com/blog/entry/1436`
9. Using haskell for competitive programming (2017). URL `https://www.reddit.com/r/haskell/comments/5sspgd/using_haskell_for_competitive_programming/`
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press (2009)
11. Erwig, M.: Functional programming with graphs. Association for Computing Machinery, New York, NY, USA (1997)
12. Erwig, M.: Inductive graphs and functional graph algorithms **11**(5) (2001)
13. Erwig, M., Iv, P.: Graph algorithms = iteration + data structures? - the structure of graph algorithms and a corresponding style of programming (1996)
14. Hughes, J.: Why Functional Programming Matters. The Computer Journal **32**(2), 98–107 (1989)
15. Hutton, G.: Programming in Haskell, 2nd edn. Cambridge University Press, USA (2016)
16. King, D.: Functional programming and graph algorithms (1996). Ph.D. Thesis
17. King, D.J., Launchbury, J.: Structuring depth-first search algorithms in haskell. Association for Computing Machinery, New York, NY, USA (1995)
18. Lipovaca, M.: Learn You a Haskell for Great Good! A Beginner's Guide, 1st edn. No Starch Press, USA (2011)
19. MichaelGilliland: Roughing out dijkstra's algorithm in haskell (2019). URL `https://www.youtube.com/watch?v=RvWnYrQ_WSk&ab_channel=MichaelGilliland`
20. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, USA (1998)
21. O'Neill, M.E., Burton, F.W.: A new method for functional arrays **7**(5) (1997)
22. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. Association for Computing Machinery, New York, NY, USA (1993)
23. Thompson, S.: The Haskell: The Craft of Functional Programming, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., USA (2011)
24. Wadler, P.: Strictness analysis aids time analysis. Association for Computing Machinery, New York, NY, USA (1988)
25. Wadler, P.: The essence of functional programming. Association for Computing Machinery, New York, NY, USA (1992)

A List of Problems

Hackerrank: (1) Solve Me First, (2) Simple Array Sum, (3) Compare the Triplets, (4) A Very Big Sum, (5) Diagonal Difference, (6) Plus Minus, (7) Staircase, (8) Mini-Max Sum, (9) Birthday Cake Candles, (10) Time Conversation, (11) Grading Students, (12) Apple and Orange, (13) Number Line Jumps, (14) Between Two Sets, (15) Break the Records, (16) Subarray Division, (17) Divisible Sum Pairs, (18) Migratory Birds, (19) Bill Division, (20) Sales by Match, (21) Counting Valleys, (22) Electronics Shop, (23) Cats and a Mouse, (24) Forming a Magic Square, (25) Picking Numbers, (26) Climbing the Leaderboard, (27) The Hurdle Race, (28) Designer PDF Viewer, (29) Utopian Tree, (30) Angry Professor, (31) Beautiful Days and the Movies, (32) Viral Advertising, (33) Save the Prisoner, (34) Circular Array Rotation, (35) Sequence Equation, (36) Find Digits, (37) Extra Long Factorials, (38) Append and Delete, (39) Sherlock and Squares, (40) Library Fine, (41) Cut the Sticks, (42) Non-Divisible Subset, (43) Repeated String, (44) Jumping on the Clouds, (45) Equalize the Array, (46) ACM ICPC Team, (47) Taum and B'day, (48) Organizing Containers of Balls, (49) Encryption, (50) Bigger is Greater, (51) Modified Kaprekar Numbers, (52) Beautiful Triplets, (53) Minimum Distances, (54) Halloween Sale, (55) The Time in Words, (56) Chocolate Feast, (57) Service Lane, (58) Lisa's Workbook, (59) Flatland Space Stations, (60) Fair Rations, (61) Cavity Map, (62) Manasa and Stones, (63) The Grid Search, (64) Happy Ladybugs, (65) Strange Counter, (66) 3D Surface Area, (67) Absolute Permutation, (68) Larry's Array, (69) Almost Sorted, (70) Intro to Tutorial Challenges, (71) HackerRank in a String, (72) Quicksort 1 - Partiion, (73) Counting Sort 1, (74) Counting Sort 2, (75) The Full Counting Sort, (76) Beautiful Binary String, (77) Closest Numbers, (78) The Love-Letter Mystery, (79) Find the Median, (80) Palindrome Index, (81) Anagram, (82) Making Anagrams, (83) Sherlock and the Valid String, (84) Sherlock and Anagrams, (85) Common Child, (86) Roads and Libraries, (87) Journey to the Moon, (88) Subset Component, (89) Breadth First Search: Shortest Reach, (90) Kruskal (MST): Really Special Subtree, (91) Even Tree, (92) Snakes and Ladders: The Quickest Way Up, (93) Prim's (MST): Special Subtree, (94) Dijkstra: Shortest Reach 2, (95) Clique, (96) Floyd: City of Blinding Lights, (97) The Coin Change Problem, (98) Them Maximum Subarray, (99) The Longest Common Subsequence, and (100) AND Product

Codewars: (1) Sum by Factors, (2) Permutations, (3) Vasya - Clerk, (4) Isograms, (5) Sum of Digits/Digital Root, (6) Credit Card Mask, and (7) Multiply

Haskeller in Code Contests

I/O

```haskell
import Control.Monad (replicateM)
import Text.Printf   (printf)

readInts :: IO [Int]
readInts = fmap (fmap read . words) getLine

main = do
  -- case 1: size first, elements next line
  -- 5
  -- 1 2 3 4 5
  _  <- getLine        -- skip a line
  ns <- readInts       -- read ints to a list
  -- case 2: num of lists, followed by lists
  -- 3
  -- 1 2 3
  -- 3 4
  -- 6 8 9 10
  n   <- readLn :: IO Int -- read 3 as int
  nss <- replicateM n readInts -- 3 lists
  -- case 3: fixed numbers on a line
  -- 8 5 7
  [height, width, weight] <- readInts
  -- case 4: variable number of structures
  -- 1
  -- john
  -- 25
  n'  <- readLn :: IO Int -- read 1 as int
  ppl <- replicateM n' $ do
    name <- getLine
    age  <- readLn :: IO Int
    return (name, age)  -- name age tuple

  -- use let to process raw inputs
  let beerSafe = filter (\(_, a)->a > 21) ppl

  -- case 1: [2, 4, 5] => 2 4 5
  putStr $ unwords (fmap show ns)
  -- case 2: [2, 4, 5] => 2\n4\n5\n
  putStr $ unlines (fmap show ns)
  -- case 3: print a single int
  print $ min 3 4
  -- case 4: good old printf
  printf "%d %d" n n'
```

Dynamic Programming

`Array` building is **lazy** so dynamic programming looks just like recursion in Haskell. Replace recursive function calls with array access.

```haskell
array :: Ix i -- Int is Ix
  => (i, i)   -- bound, e.g. (1, 5)
  -> [(i, e)] -- value e at index i
  -> Array i e

-- recursive fib 30: 1.62 sec, 574 MB
-- DP fib 30: 0.00 sec, 92 KB
fib n = go n
 where
  go 1 = 1
  go 2 = 1
  go m = arr ! (m-1) + arr ! (m-2)
  arr  = listArray (1,n) [go i | i <- [1..n]]
```

2-D Example: *Coin change*, given a list of coin denominations `ns`, calculate how many ways are there to change `m` cents.

Example: `m = 10, ns = [2,5,3,6]`

	0	1	2	3	4	5	6	7	8	9	10
2	1	0	1	0	1	0	1	0	1	0	1
5	1	0	1	0	1	1	1	1	1	1	2
3	1	0	1	1	1	2	2	2	3	3	4
6	1	0	1	1	1	2	3	2	4	4	5

```haskell
-- Answer is: c ! ((length ns), m)
-- as is the array copy of ns
as   = listArray (1,n) ns
bnds = ((0,0), (length ns, m))
c    = listArray bnds
               [go i j | (i,j) <- range bnds]
go i j | i == 0          = 0
       | j == 0          = 1
       | j - (as!i) < 0  = c ! (i-1,j)
       | otherwise       = c ! (i-1,j) +
                           c ! (i,j-(as!i))
```

Kruskal MST

```haskell
-- disjoint set
-- union -> union
-- find  -> partition with (u or v)
type DSet = [IntSet]
-- labelled edage
type Edge = (Int, (Int, Int))

kruskal :: Set Edge -> DSet -> Int -> Int
kruskal pq ds sum
  | Set.null pq    = sum
  | [] <- uv       =
      kruskal pq' (fromList [u,v]:ys) (sum+w)
  | [s1, s2] <- uv =
      kruskal pq' (union s1 s2 : ys) (sum+w)
  | otherwise = kruskal pq' ds sum
  where ((w,(u,v)),pq') = Set.deleteFindMin pq
        (uv, ys)        =
          partition
            ((||) <$> member u <*> member v)
            ds
```

Debugging

ghci debugger. You have access to *free variables* of the expression to be evaluated.

Command	Description
`:break <lineno>`	Set break point
`:list`	Print code
`main`	Run code
`:show breaks`	Show break points
`:show context`	Show variable binding
`:set stop <action>`	Do action when a break point is hit
`:step`	Step
`:print x`	Print the value of x, forces x if was a thunk
`:hist`	History of evaluation steps
`:abandon`	Stop execution
`:set +s`	Measure running time and memory used

Debug.Trace

`trace` and `traceShow`

Graph Representations

```haskell
import Control.Monad
import Data.Array    (Array, accumArray)
import Data.List     (foldl)
import Data.Set      (empty, insert, toList)

readInts :: IO [Int]
readInts = fmap (fmap read . words) getLine

-- unlabelled graph representation
type Vertex = Int
type Graph  = Array Vertex [Vertex]

-- intermediate representation for dfs
data Tree a   = Node a [Tree a]
type Forest a = [Tree a]

-- intermediate representation for bfs
type LNode  = Vertex
type LPath  = [LNode]
type LRTree = [LPath]

-- build an unlabelled graph
buildG :: Int -- number of vertices
       -> [(Vertex, Vertex)] -- edges
       -> Graph
buildG n es =
  accumArray
    (flip (:)) -- accumulate dest to list
    [] -- each adjacent list starts empty
    (1, n) -- vertex number from 1 to n
    (es ++ fmap revE es) -- bidirectional
  where revE (x, y) = (y, x)

-- read in a graph
main = do
  [n, m] <- readInts -- n nodes, m edges
  es     <- replicateM m readInts
  let nodup = -- remove duplicates in set
        foldl (flip insert)
              empty $
              fmap (\[x,y] -> (x,y)) es
      g     = buildG n (toList nodup)
  -- if using Data.Graph connected components
  -- to get number of components in g
  print $ length (components g)
```

Graph: Dijkstra

```haskell
type Node   = Int
type Weight = Int
type Graph  = Array Node [(Node, Weight)]

type LNode    = (Node, Weight)
newtype LPath = LPath [LNode] deriving Show
type LRTree   = [LPath]

instance Eq LPath where
  (LPath ((_,w):_)) == (LPath ((_,v):_)) = w == v

instance Ord LPath where
  (LPath ((_, w):_)) `compare`
    (LPath ((_, v):_)) = w `compare` v

mkEdge (s,t,w)   = (s, (t, w))
revE (s, (t, w)) = (t, (s, w))

buildG :: Int -> [(Int,Int,Int)] -> Graph
buildG n edges =
  accumArray (flip (:)) [] (1,n) biEdges
  where
    biEdges = es ++ fmap revE es
    es      = mkEdge <$> edges

dijkstra :: Int -> Int -> Graph -> LRTree
dijkstra n s =
  dijkstra'(S.fromList [1..n])
           (singleton (LPath [(s, 0)]))

dijkstra' :: IntSet -> Heap LPath
                    -> Graph -> LRTree
dijkstra' vs h g
  | isEmpty h ||
    S.null vs        = [] -- Data.Set as S
  | S.notMember v vs = dijkstra' vs h' g
  | otherwise        = p:dijkstra' vs' h'' g
  where
    (p@(LPath ((v,w):_)), h') = findDeleteMin h
    vs'  = delete v vs
    hs   = expand w p (g ! v)
    h''  = roll (h':hs)

expand :: Int -> LPath -> [(Node, Weight)]
               -> [Heap LPath]
expand d (LPath p) = fmap
  (\(w, d') -> singleton (LPath ((w, d+d'):p)))

-- Heap is from [Okasaki 98] page 198
```

More Data Types

Use these "special" ones only when you know regular `Int` or `String` will slow down the program.

Type	Description
`Int`	signed 30-bit, most likely 32 or 64 implementations
`Int8`	8-bit signed. Others include `Int16`, `Int32`, `Int64`
`Word8`	8-bit unsigned. Others include `Word16`, `Word32`, `Word64`
`Integer`	Arbitary-precision integers
`ByteString`	Represents sequences of bytes or 8-bit characters. More efficient but has the same interfaces as `String`. Used qualified import.

Bitwise Operation are supported through the `Bits` type class. Most integral types are instances of it.

```haskell
import Data.Bits

5 .&. 3    -- bitwise and
ShiftR 5 3 -- shift right 3 places
```

Integer to binary (as String). The last `""` is needed by `ShowS` which uses a *difference list* trick for efficiency.

```haskell
import Data.Char (intToDigit)
import Numeric   (showIntAtBase)

showIntAtBase 2 intToDigit 12 "" -- 1100
```

Campanion Paper

To see how the choices were made and related work: TBA