Pure and Lazy Lambda Mining
An Experience Report

Nicolas Wu[1(✉)], José Pedro Magalhães[1(✉)], Jeroen Bransen[2(✉)], and Wouter Swierstra[2(✉)]

[1] Department of Computer Science, University of Oxford, Oxford, UK
{nicolas.wu, jose.pedro.magalhaes}@cs.ox.ac.uk
[2] Department of Computer Science, Utrecht University, Utrecht, The Netherlands
{j.bransen, w.s.swierstra}@uu.nl

Abstract. This paper discusses our entry to the 2012 ICFP Programming Contest, written entirely in Haskell. Our solution uses many features of Haskell: pure immutable data structures, laziness, higher-order functions, concurrency, and exception handling. Each of these features plays an essential part in our overall solution, and we demonstrate how these key elements can be composed together. In this exposition, we stress the importance of how the code was structured in such a way that made safely refactoring and extending the model a relatively easy task, and how Haskell's strong type system made it possible for our team to remain agile under changing specifications.

1 Introduction

In the classic paper *Why Functional Programming Matters*, Hughes [3] argues that functional programming in Miranda provides two kinds of glue that enable the modular construction of programs: lazy evaluation and higher order functions. To drive this point home, Hughes presents several small and elegant example programs that rely on precisely these features. But how useful are laziness and higher order functions in larger developments?

This paper investigates this question and aims to provide further evidence supporting Hughes's claim. We describe a solution to the 2012 ICFP programming contest.[1] This programming contest allows participants to write solutions in any language, or combination of languages, in a time frame of 72 hrs. Our solution was entirely implemented in Haskell [4]. We describe our solution as it was developed in the 72 hours of the contest, plus some later refactoring for readability and bug fixing. Crucially, the solution we present uses many different Haskell features: pure immutable data structures, laziness, higher-order functions, concurrency, and exception handling.

Nicolas Wu and José Pedro Magalhães have been funded by EPSRC grant number EP/J010995/1.

[1] The official task description is available at http://icfpcontest2012.wordpress.com/task/. A video presenting the task and announcing the winners can be seen at https://www.youtube.com/watch?v=5TCqUU3-GT0.

R. Hinze (Ed.): IFL 2012, LNCS 8241, pp. 207–223, 2013.
DOI: 10.1007/978-3-642-41582-1_13,

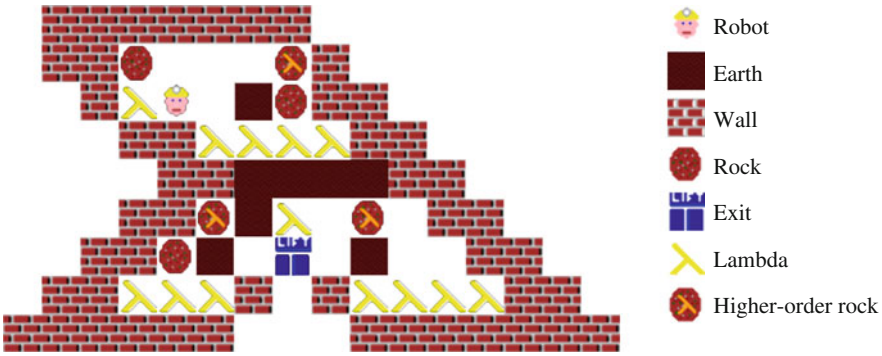


Fig. 1. Graphical representation of a mine

1.1 Problem Description

The ICFP programming contest has been run every year since 1998. This year, participants were invited to program a virtual mining robot to collect resources called 'lambdas' while avoiding falling rocks, getting trapped, or drowning. The overall score of a route was determined by the number of lambdas collected and the number of moves required to collect those lambdas. Figure 1 shows a graphical depiction of a game in progress. The goal is to compute a sequence of moves for the robot to collect as many lambdas as possible, without being crushed by falling rocks. If all the lambdas are collected, reaching the exit gives an extra score bonus.

The problem specification was extended four times over the course of the competition, demanding efficient and correct code to be produced under tight deadlines. This provided an excellent means of substantiating the claim that functional programming languages help to produce code that is both modular and reusable. In the remainder of this paper, we describe our solution and how it relies on several key Haskell features. The precise description of the problem will become clear from the presentation of our solution.

We begin by describing pure models of both the mine (Sect. 2) and the search space (Sect. 3). Our solution uses a combination of search strategies (Sect. 4), that traverse the shared search space. The main program then applies these strategies in parallel (Sect. 5), returning the best result. Section 6 describes the changes necessary to adapt our solution to each of the problem specification extensions. We conclude in Sect. 7 with a summary of our experience, including a number of practical guidelines for code development in a situation similar to ours.

2 Pure Modelling

In this section we describe how we model and simulate the problem in Haskell.

2.1 Model

The model represents the entire state of a mine at any given time, and forms an important interface for the rest of the system: the simulator (Sect. 2.2) takes one state of the model to the next, the parser must produce a value of this type, the visualiser outputs a visual rendering of the model (Sect. 2.3), and various strategies can be employed based on the state held within the model (Sect. 4).

The basic building block of a mine is a *Tile*, which holds information about what exists at a particular coordinate:

$$\textbf{data}\ \mathit{Tile} = \mathit{Robot} \mid \mathit{Wall} \mid \mathit{Rock}\ \mathit{Bool} \mid \mathit{Lambda} \mid \mathit{Earth} \mid \mathit{Empty} \mid \mathit{Exit}$$

Note that rocks are parameterised by a Boolean which indicates whether or not a rock is falling: when the robot is directly beneath a falling rock, it is crushed.

Each tile in the mine is given a specific coordinate, which is simply a pair of *Int* values named *Coord*. Putting these elements together, we are interested in an array that is indexed by *Coord*s and contains *Tiles*. This describes the layout of the mine:

$$\textbf{type}\ \mathit{Layout} = \mathit{Array}\ \mathit{Coord}\ \mathit{Tile}$$

Using an array for this representation is appropriate, since we need to perform lookups of elements at coordinates very often, and arrays have constant time lookup.

It is useful to define a function that checks the value of a tile in the layout at a particular coordinate, by dereferencing the appropriate location in the array:

$$\begin{aligned}&\mathit{isTile} :: \mathit{Layout} \to \mathit{Coord} \to \mathit{Tile} \to \mathit{Bool}\\ &\mathit{isTile}\ l\ xy\ t = l\,!\,xy \equiv t\end{aligned}$$

There is an important caveat to using this function and others like it which make use of (!), the unsafe indexing operator. This operator is efficient, but makes no effort to ensure that the coordinates being sought are within the bounds of the array, and this is a danger which could easily result in an exception being thrown at runtime.

Another utility function finds the coordinates of all the tiles which satisfy a given predicate:

$$\begin{aligned}&\mathit{findTiles} :: (\mathit{Tile} \to \mathit{Bool}) \to \mathit{Layout} \to [\mathit{Coord}]\\ &\mathit{findTiles}\ p = \mathit{map}\ \mathit{fst} \circ \mathit{filter}\ (p \circ \mathit{snd}) \circ \mathit{assocs}\end{aligned}$$

This works by getting a list of all the associations in the array and representing these as a value of type $[(\mathit{Coord}, \mathit{Tile})]$. This list is then filtered by the predicate, before the coordinates are extracted.

While the *Layout* structure holds much of the information required during the game, some essential features are lacking, such as the number of moves that have passed since the beginning of the game. The whole state is saved in a structure named *Mine*, which contains all the information required for assessing the current score:

$$\begin{aligned}&\textbf{data}\ \mathit{Mine} = \mathit{Mine}\ \{\,\mathit{layout} &&:: \mathit{Layout}\\ &\qquad\qquad\quad\ \ ,\,\mathit{robot} &&:: \mathit{Coord}\\ &\qquad\qquad\quad\ \ ,\,\mathit{lambdas} &&:: \mathit{Int}\\ &\qquad\qquad\quad\ \ ,\,\mathit{moves} &&:: \mathit{Int}\,\}\end{aligned}$$

In particular, *Mine* stores the current position of the robot along with the number of remaining lambdas and the number of moves it has taken to reach this point, since this is an important part of calculating the score.

When the robot has finished collecting all the lambdas, the exit opens and the robot is allowed to leave the mine. Our representation indicates that the robot has exited when the robot's coordinates correspond with the *Exit* tile in the layout:

$$\begin{aligned}&\mathit{isDone} :: \mathit{Mine} \to \mathit{Bool}\\ &\mathit{isDone}\ \mathit{mine} = \mathit{isTile}\ (\mathit{layout}\ \mathit{mine})\ (\mathit{robot}\ \mathit{mine})\ \mathit{Exit}\end{aligned}$$

The task of ensuring that the robot can only enter an exit when all lambdas have been collected is left to the simulator, which we explain in the next section.

2.2 Simulation

The simulation code determines how the system responds to the robot's actions: each time the robot makes a move the world is updated and a new *Mine* value is calculated.

The robot can perform several moves: moving up, down, left, right, waiting, or aborting the mission. For brevity, the data constructors that represent these moves contain only the initial letter of each action:

$$\textbf{data}\ \mathit{Move} = L \mid R \mid D \mid U \mid W \mid A$$

We often calculate coordinates based on a sequence of moves; the following function returns a coordinate that has been shifted by some movement value:

$$\begin{aligned}&(\rightsquigarrow) :: \mathit{Coord} \to \mathit{Move} \to \mathit{Coord}\\ &(x, y) \rightsquigarrow L\ \ = (x - 1, y\ \ \ \ )\\ &(x, y) \rightsquigarrow R\ \ = (x + 1, y\ \ \ \ )\\ &(x, y) \rightsquigarrow D\ \ = (x\ \ \ \ , y - 1)\\ &(x, y) \rightsquigarrow U\ \ = (x\ \ \ \ , y + 1)\\ &(x, y) \rightsquigarrow \_\ = (x\ \ \ \ , y\ \ \ \ )\end{aligned}$$

For example, this operator is used to verify whether the robot has been crushed by a rock, which happens whenever the tile directly above the robot is a falling rock:

$$\begin{aligned}&\mathit{isDead} :: \mathit{Mine} \to \mathit{Bool}\\ &\mathit{isDead}\ \mathit{mine} = \mathit{isRockFalling}\ (\mathit{layout}\ \mathit{mine}\,!\,(\mathit{robot}\ \mathit{mine} \rightsquigarrow U))\end{aligned}$$

The function *isRockFalling* distinguishes rocks that are falling.

The score is calculated by multiplying a constant factor per collected lambda minus the number of moves the robot made. The constant depends on how the game ended, and is 75 when all lambdas were collected, 25 when the robot dies, and 50 if the robot aborted (which is the default action when no more moves are made).

The central function used to simulate the robot's progression through a mine is *step*, which takes a current mine, a move, and steps the simulator through that move:

```
step :: Mine → Move → Mine
step mine A      = mine
step mine move = mine' where
  (layout', robot') = stepRobot mine move
  layout''          = array ((bounds ∘ layout) mine) $
    concat [ updRocks (mine {layout = layout'}) (x, y) (layout' ! (x, y))
           | y ← [1 .. h], x ← [1 .. w]]
  moves'            = 1 + moves mine
  lambdas'          | isTile (layout mine) robot' Lambda = lambdas mine − 1
                    | otherwise                         = lambdas mine
  (w, h)            = (snd ∘ bounds ∘ layout) mine
  mine'             = mine {layout  = layout''  , robot   = robot'
                            , lambdas = lambdas', moves = moves'}
```

When a move other than A is requested, the simulator returns the result of the updated record *mine'*. The *layout* field is updated in two stages. First the value of the layout is calculated after the robot has made its step and stored in *layout'*, and then this value is used in creating a new array, *layout''*, that contains the state of the mine after all the falling of rocks has been calculated. This follows the problem specification.

Updating the robot is left to the *stepRobot* function, which returns the layout after the robot has moved, and gives the new coordinate of the robot:

```
stepRobot :: Mine → Move → (Layout, Coord)
stepRobot mine move =
  case l ! xy' of
    Earth      → (l // [(xy', Robot), (xy, Empty)], xy')
    Empty      → (l // [(xy', Robot), (xy, Empty)], xy')
    Lambda     → (l // [(xy', Robot), (xy, Empty)], xy')
    Exit       | lambdas mine ≡ 0
               → (l // [(xy', Robot), (xy, Empty)], xy')
    Rock _     | (move ≡ L ∨ move ≡ R) ∧ isTile l (xy' ⇝ move) Empty
               → (l // [(xy', Robot), (xy, Empty), (xy' ⇝ move, Rock False)], xy')
               → (l // [(xy, Robot)], xy)
    _
  where l  = layout mine
        xy = robot mine
        xy' = xy ⇝ move
```

Moving towards earth, an empty tile, or a lambda simply updates the robot position, leaving an empty space behind. Moving towards the exit is only allowed if all the lambdas have been collected. Moving towards a rock is possible if the movement is sideways, and there is empty space next to the rock being pushed. All other movements are invalid, and the robot remains in the same position.

Another crucial function is *updRocks*, which is responsible for updating the position of rocks after the robot has moved:

$$
\begin{aligned}
&\mathit{updRocks} :: \mathit{Mine} \to \mathit{Coord} \to \mathit{Tile} \to [(\mathit{Coord}, \mathit{Tile})] \\
&\mathit{updRocks}\ \mathit{mine}\ \mathit{xy}\ (\mathit{Rock}\ \_) \\
&\quad \mid \mathit{isFallDown}\ \ l\ \mathit{xy} = [(\mathit{xy}, \mathit{Empty}), (\mathit{xy} \rightsquigarrow D\ \ \ \ \ \ , \mathit{Rock}\ \mathit{True})] \\
&\quad \mid \mathit{isFallRight}\ \ l\ \mathit{xy} = [(\mathit{xy}, \mathit{Empty}), (\mathit{xy} \rightsquigarrow D \rightsquigarrow R, \mathit{Rock}\ \mathit{True})] \\
&\quad \mid \mathit{isFallLeft}\ \ \ l\ \mathit{xy} = [(\mathit{xy}, \mathit{Empty}), (\mathit{xy} \rightsquigarrow D \rightsquigarrow L, \mathit{Rock}\ \mathit{True})] \\
&\quad \mid \mathit{isFallLambda}\ l\ \mathit{xy} = [(\mathit{xy}, \mathit{Empty}), (\mathit{xy} \rightsquigarrow D \rightsquigarrow R, \mathit{Rock}\ \mathit{True})] \\
&\quad \mid \mathit{otherwise}\ \ \ \ \ \ \ \ \ = [(\mathit{xy}, \mathit{Rock}\ \mathit{False})] \\
&\quad \textbf{where}\ l = \mathit{layout}\ \mathit{mine} \\
&\mathit{updRocks}\ \_\ \mathit{xy}\ \mathit{tile} = [(\mathit{xy}, \mathit{tile})]
\end{aligned}
$$

The functions *isFallDown*, *isFallRight*, *isFallLeft*, and *isFallLambda* determine whether the rock will fall in a particular direction. These are all predicates that take a *Layout* and a *Coord*, and simply output the appropriate *Bool*.

Keeping the entire state of a mine as a single value of type *Mine* enables the definition of *step* to remain relatively simple, since all of the required data for an update is held in a single structure. This complete encapsulation of state means that there are no implicit outside dependencies to handle when trying to evaluate a particular mine.

2.3 Input and Output

The input maps are supplied in text format. To read these into our model, we wrote a text parser using Attoparsec,[2] working on *ByteString*s for efficiency reasons. The input format is simple, so the parser is unsurprising and therefore omitted in this presentation.

Visualising the maps in a user-friendly way was not a requirement of the contest. However during development it was helpful to visualise maps and generated solutions, and to be able to manually play each mine. Due to time considerations we developed only a simple ANSI text-based visualiser, which was enough for our testing purposes.

3 The Game Trie

One of the key benefits of Haskell is its purity, allowing game states to be shared across different solvers. Our strategy for exploiting this was to spawn a number of different agents that explore a shared data structure that holds paths to different game states together with their scores.

[2] http://hackage.haskell.org/package/attoparsec

3.1 Tries

The structure we use to encode paths through the mine is a non-empty trie [2]:

```
data Trie k v = Trie {root :: v, branches :: Map k (Trie k v)}
```

An important aspect of a value of type *Trie k v* is that it can behave like a map of type *Map [k] v*, and this forms the basis of an intuitive interface with a number of well-understood standard functions. These standard functions on *Trie* will prove useful in the strategy code (Sect. 4), since the entire search space of a game can be encoded as a trie, mapping sequences of moves to a game state:

```
type GameTrie  = Trie Move GameState
data GameState = GameState {gameStateMine :: Mine
                           , gameStateScore :: Score}
```

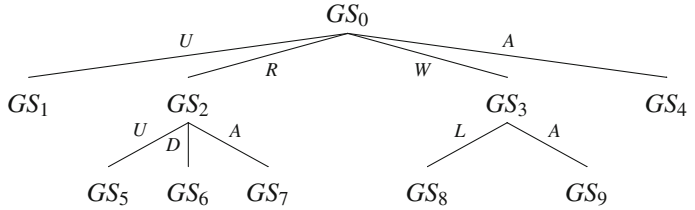
For instance, we can lookup the *GameState* associated with a certain path by using the familiar *lookup* function:

```
lookup :: (Eq k, Ord k) ⇒ [k] → Trie k v → Maybe v
lookup []       (Trie v _)    = Just v
lookup (k : ks) (Trie _ kvs) = Map.lookup k kvs ≫= lookup ks
```

A *Path* is represented by a list of moves:

```
type Path = [Move]
```

The type *GameTrie* operates much like the type of *Map Path GameState*, but its encoding is very efficient; each branch of the tree encodes one possible move, as illustrated in the following figure:



In this example, starting from some initial game state GS_0, the robot can move up and die, resulting in game state GS_1, with no further paths. Alternatively, the robot can go right, and then proceed either up, down, or abort. A *GameTrie* is computed by starting with an initial state (of score zero), and considering only valid moves from the current position:

```
mkTrie :: (Eq k, Ord k) ⇒ v → (v → [k]) → (v → k → v) → Trie k v
mkTrie v f next = Trie v (Map.fromList [(k, mkTrie (next v k) f next) | k ← f v])
gameTree :: Mine → GameTrie
gameTree mine0 = mkTrie (GameState mine0 0 (hash mine0))
                        (goodMoves ∘ gameStateMine)
                        (mkGameState mine0 ∘ gameStateMine)
```

We omit the function *mkGameState*, which simply computes the current *GameState*, and function *goodMoves*, which returns the valid moves for the robot. One of the key features of our solution is that the *GameTrie* represents all the paths in the mine, and this trie is shared over the different robot strategy algorithms. This means that states are never computed twice; if strategy one already went down a particular path, the next strategy can immediately get the corresponding game state for that path, without having to step through each move. In addition, equivalent states that are reachable through different paths are not recomputed, and this is achieved through the use of hashes, described in more detail in Sect. 3.3.

Another useful property of values of type *Trie k v* is that they behave like trees of type *Tree* $([k], v)$, which brings another family of standard functions that are well understood. In particular, a tree can be traversed in breadth-first order in order to compute all possible paths in increasing length:

```
flatten :: Trie k v → [([k], v)]
flatten = concat ∘ levels

levels :: Trie k v → [[([k], v)]]
levels tree = (map extract ∘ iterate expand) [([], tree)]
  where
    expand :: [([k], Trie k v)] → [([k], Trie k v)]
    expand = concatMap (λ(sk, Trie _ kts) → map (first (:sk)) (Map.toList kts))
    extract :: [([k], Trie k v)] → [([k], v)]
    extract = map (λ(sk, Trie v' _) → (reverse sk, v'))
```

In Sect. 3.2 we will use variations of these functions to build efficient pathfinding algorithms that are used to search for solutions within the *GameTrie*.

3.2 Pathfinding

The key to our strategy is to navigate the *Trie* structure, and identify a path that leads to a high score. The following function, for example, finds the paths to the exit:

```
solve :: Mine → [(Path, GameState)]
solve mine = (filter (isDone ∘ gameStateMine ∘ snd) ∘ flatten ∘ gameTree) mine
```

Since *flatten* produces a breadth first traversal of the tree, we know that the result at the head of the list will have the shortest path. Furthermore, since the predicate applied is *isDone*, we know that the solution found is for a completed mine. Therefore, the head of this list will contain a solution with the maximal score for a completed mine!

However, while this strategy would eventually find such a solution for completable mines, it is prohibitively inefficient. In addition, since the tree is potentially very large, and not all mines are necessarily completable, an exhaustive search will generally not be possible. In order to solve this, we break the problem

down into finding paths to a number of intermediate states given by some predicate: the basis for the searches will still be variations on breadth first search, but the goal is different. Rather than finding paths to different values of type *GameState*, we will seek values of type *GameTrie*, so that we can search for new paths based on the returned tree, thus giving us more sophisticated searching strategies, where intermediate goals are reached and further analysis is performed on the trees that follow on from the paths to those goals.

A useful utility function along these lines is *findPaths*, which looks for paths to a particular coordinate:

$$\begin{aligned}&\mathit{findPaths} :: \mathit{GameTrie} \to \mathit{Coord} \to [(\mathit{Path}, \mathit{GameTrie})]\\&\mathit{findPaths}\ \mathit{tree}\ \mathit{dest} = \mathit{bfs}\ ((\equiv)\ \mathit{dest} \circ \mathit{robot} \circ \mathit{gameStateMine})\ \mathit{tree}\end{aligned}$$

This can be used, for example, to find a path to the *Exit* once the task of collecting all the lambdas is complete:

$$\begin{aligned}&\mathit{findExits} :: \mathit{GameTrie} \to [(\mathit{Path}, \mathit{GameTrie})]\\&\mathit{findExits}\ \mathit{tree} = \mathit{findTiles}\ (\equiv \mathit{Exit})\ (\mathit{layout}\ (\mathit{getMine}\ \mathit{tree})) \gg\!= \mathit{findPaths}\ \mathit{tree}\end{aligned}$$

This works by first finding the appropriate tile, and, if such a coordinate is found, then it is used by *findPath* to calculate a path.

At the heart of *findExits* is an efficient breadth first search algorithm, with a more general interface than that of *solve*. A naive breadth first search that operates on the *Trie* structure can be described as follows:

$$\begin{aligned}&\textbf{type}\ \mathit{KTrie}\ k\ v = ([k], \mathit{Trie}\ k\ v)\\&\mathit{bfsNaive} :: (v \to \mathit{Bool}) \to \mathit{Trie}\ k\ v \to [\mathit{KTrie}\ k\ v]\\&\mathit{bfsNaive}\ p\ \mathit{tree} = (\mathit{filter}\ (p \circ \mathit{root} \circ \mathit{snd}) \circ \mathit{stems})\ [([], \mathit{tree})]\end{aligned}$$

This makes use of the function *stems*, which is similar to *flatten*, but returns a list of paths with corresponding subtrees:

$$\begin{aligned}&\mathit{stems} :: [\mathit{KTrie}\ k\ v] \to [\mathit{KTrie}\ k\ v]\\&\mathit{stems}\ [\,] = [\,]\\&\mathit{stems}\ ((\mathit{sk}, t@(\mathit{Trie}\ \_\ \mathit{kts})) : \mathit{skts}) = (\mathit{reverse}\ \mathit{sk}, t) : \mathit{stems}\ \mathit{skts}'\\&\quad \textbf{where}\ \mathit{skts}' = \mathit{skts} +\!\!+ [(k' : \mathit{sk}, t') \mid (k', t') \leftarrow \mathit{Map.toList}\ \mathit{kts}]\end{aligned}$$

The *stems* function produces a breadth-first traversal of the tree, but is certainly not optimal: this function makes no effort to ensure that some common state has not been investigated several times: certain paths lead to exactly the same state, and we have no reason to assume that there will be any implicit sharing of these states.

3.3 Hashing

During the lazy construction of the tree structure, sharing is not exploited between nodes that are equal. As a result, a search of the tree will likely result

in repeated inspections of equal nodes and their children: this happens whenever there is more than one path to a particular state. To avoid this expensive recomputation, the breadth first search algorithm is modified to contain an accumulator that keeps track of the nodes visited so far, and will not queue nodes whose values have already been visited elsewhere.

Rather than have the accumulator store the entire state of each visited mine, and have to perform an expensive equality operation, a hash of the mine is stored instead. We therefore extend the type of a *GameState* so that it contains a *Hash*:

```
type Hash = Int
data GameState = GameState { ...
                           , gameStateHash :: Hash }
```

An instance of *Hashable* is provided, giving us a means of obtaining the hash of a *Mine*:

```
instance Hashable Mine where
  hash mine = hash ((hash ∘ assocs ∘ layout) mine
                   ,(hash ∘ robot) mine
                   ,(hash ∘ moves) mine)
```

An accumulator, which is a set of hashes, is then added to the machinery of *stems* that allows states which have already been visited to be pruned:

```
stemsPrune :: Hashable v ⇒ Set Hash → [KTrie k v] → [KTrie k v]
stemsPrune _       []                              = []
stemsPrune visited ((sk, t@(Trie v kts)) : skts) = case insertM (hash v) visited of
  Nothing       → stemsPrune visited skts
  Just visited' → (reverse sk, t) : stemsPrune visited' skts'
  where skts' = skts ++ [(k' : sk, t') | (k', t') ← Map.toList kts]
insertM :: Ord a ⇒ a → Set a → Maybe (Set a)
insertM x xs | Set.member x xs = Nothing
             | otherwise       = Just (Set.insert x xs)
```

The idea is to keep an accumulator that checks if the value of the tree being examined has been visited before. If it has been visited, then this value is rejected by the function *insertM*, and the next candidate for traversal is considered. If the value has not yet been visited, then the tree that contains it is added to the output of the search, its content is added to the set of visited values, and children are scheduled for traversal.

This lets us define a breadth first search that does not visit the same subtree twice:

```
bfsPrune :: Hashable v ⇒ (v → Bool) → Trie k v → [KTrie k v]
bfsPrune p t = filter (p ∘ root ∘ snd) ∘ stemsPrune Set.empty $ [([], t)]
```

The beauty of this solution is that it requires only the values v of the *Trie k v* structure to be *Hashable*. However, this does not come without its cost: the hashing itself is not perfect, and so it is possible that two different states hash to the same value. If this were to happen, then not all unexplored states will be visited, since we would incorrectly discard states that collide with already visited states that a hash. In practice, this does not turn out to pose a problem, since the hash space is large enough.

Another performance issue is that *stems* uses a list to hold the queue of subtrees left to visit: the performance of appending to the end of a list is poor, and this can be easily improved by using a queue structure instead, and replacing the call to *stemsPrune* with an adequately instantiated call to *stemsPruneQ*.

```
stemsPruneQ :: Hashable v ⇒ Set Hash → Seq (KTrie k v) → [KTrie k v]
stemsPruneQ visited q = case Seq.viewl q of
  Seq.EmptyL                → []
  (sk, t@(Trie v kts)) :< q' → case insertM (hash v) visited of
    Nothing       → stemsPruneQ visited q
    Just visited' → (reverse sk, t) : stemsPruneQ visited'
      (foldr (flip (|>)) q' [(k' : sk, t') | (k', t') ← Map.toList kts])
bfsPruneQ :: Hashable v ⇒ (v → Bool) → Trie k v → [KTrie k v]
bfsPruneQ p t = (filter (p ∘ root ∘ snd) ∘ stemsPruneQ Set.empty ∘ return) ([], t)
```

This is a relatively straight-forward transliteration of the list based version into one that uses a *Seq* datastructure instead.

On a final note about pathfinding, the *findPaths* function takes a destination coordinate as an argument, and filters out the results of a breadth-first traversal until a state is found where the robot is at the coordinate. A heuristic for possibly improving the search is by using a distance metric which determines how close a given point is to the destination, and using this information to give priority to certain elements within the queue. This is the basis of the well known A* algorithm [1], which is widely used in path finding and graph traversal.

To implement this algorithm, much of the structure present in *bfsPruneQ* can be reused, where *Seq* is replaced by a *MinQueue* structure which orders the elements according to some comparison function. For brevity, these details are omitted, but the development revolves around choosing an appropriate comparison function: a valid option would be to use the well-known Manhattan distance between two points, although there are other possible options. This function is then used to form the priorities of elements within the *MinQueue*, which arranges its elements so that those which are closest to the destination are favoured when considering the next value to explore in the search.

4 Robot Strategy

Our solution relies on using a portfolio of simple strategy algorithms competing for finding the best solution. A strategy takes a *GameTrie* and computes possible paths through the mine, together with their score:

```
type Strategy = GameTrie → [(Path, Score)]
```

We can now write a variation of the *solve* function (from Sect. 3) that produces a *Strategy* using *bfsPruneQ*:

```
solveS :: Strategy
solveS = map (second getScore) ∘ bfsPruneQ (const True)
```

This encodes the strategy of trying all possible paths, in a breadth-first manner. Naturally, this strategy is not very efficient, and will only work on very small maps. We also have a variant strategy that looks ahead only a number steps, and then takes one step along the best path found so far. This strategy finds locally optimal solutions.

An alternative strategy orders the remaining lambdas, tries to reach each one of them, and then walks towards the exit:

```
cmpS :: Comparison → Strategy
cmpS cmp tree
  | lambdas (getMine tree) ≡ 0 = case listToMaybe $ findExits tree of
      Just (p, tree′) → [(p   , getScore tree′)]
      Nothing         → [([A], getScore tree )]
  | otherwise = case pathToLambda cmp tree of
      []                → [([A], getScore tree)]
      ((p, tree′): _) → (p, getScore tree′) : map (first (p++)) (cmpS cmp tree′)
```

We omit functions *getMine* and *getScore*, which are simple accessors of the *GameTrie* data structure. Function *pathToLambda* takes a ranking function for lambdas and returns a list of paths:

```
pathToLambda :: Comparison → GameTrie → [(Path, GameTrie)]
pathToLambda cmp tree = concatMap snd (sortBy cmp dests)
  where dests = map (λcoord → (coord, findPaths tree coord))
                    (findTiles (≡ Lambda) ((layout ∘ getMine) tree))
```

We can now define multiple strategies simply by instantiating the comparison function of *cmpS*:

```
eqCmpS, lowCmpS, highCmpS :: Strategy
eqCmpS   = cmpS (λ _ _ → EQ)
lowCmpS  = cmpS (cmpCoords (λ(_, y) (_, y′) → compare y  y′))
highCmpS = cmpS (cmpCoords (λ(_, y) (_, y′) → compare y′ y))
```

Strategy *eqCmpS* treats all lambdas equally, while *lowCmpS* prefers lambdas located the lowest in the mine. This strategy might make sense when the lower parts of the mine become harder to access as time goes by (see Sect. 6.1).

We also have more complicated strategies involving *cmpS*, such as preferring lambdas that are part of large clusters.

5 Concurrency and Exception Handling

Strategies turn the representation of a game tree into a list of paths with their corresponding score. By sharing the game tree structure, a number of concurrent worker threads using different strategies can compete with one another to find an optimal solution. The communication between these threads occurs through the use of Haskell's *MVar* values: these are mutable variables which can be shared and synchronised between threads. Initially, a trivial solution is put in *mvBest*. The task of each worker is to improve this solution with whatever they might encounter in their list of candidate answers.

$$
\begin{aligned}
&\mathit{improve} :: (\mathit{Ord}\ s, \mathit{NFData}\ s, \mathit{NFData}\ a) \Rightarrow \mathit{MVar}\ (a, s) \rightarrow [(a, s)] \rightarrow \mathit{IO}\ () \\
&\mathit{improve}\ \mathit{mvBest} = \mathit{mapM\_}\ (\lambda x \rightarrow x\ \text{`}\mathit{deepseq}\text{`}\ \mathit{modifyMVar\_}\ \mathit{mvBest}\ (\mathit{cmpBest}\ x)) \\
&\quad \textbf{where}\ \mathit{cmpBest}\ x\ \mathit{best} = \mathit{return}\ (\textbf{if}\ \mathit{snd}\ x > \mathit{snd}\ \mathit{best}\ \textbf{then}\ x\ \textbf{else}\ \mathit{best})
\end{aligned}
$$

Here, each solution x is a tuple of type (s, a), where s is a score that will be maximised, and a the answer itself. We require s and a to have an *NFData* instance to be able to force evaluation using *deepseq*, since the entire computation of the value of x should occur before blocking on the *mvBest* variable. The *MVar* is a reference to the best solution found so far; *improve* updates this *MVar* whenever a better solution is found. As this worker might be interrupted before the list is fully evaluated, it is important that *modifyMVar_* is an atomic operation: if the worker raises an exception while it is modifying *mvBest*, then the value is restored to its original state.

The workers are spawned by *spawnWorkers*, which creates a new asynchronous thread for each of the answers returned by the strategies, and then waits for all the threads to finish.

$$
\begin{aligned}
&\mathit{spawnWorkers} :: (\mathit{Ord}\ s, \mathit{NFData}\ s, \mathit{NFData}\ a) \Rightarrow \mathit{MVar}\ (a, s) \rightarrow [[(a, s)]] \rightarrow \mathit{IO}\ () \\
&\mathit{spawnWorkers}\ \mathit{mvBest}\ \mathit{xss} = \textbf{do}\ \mathit{workers} \leftarrow \mathit{mapM}\ (\mathit{async} \circ \mathit{improve}\ \mathit{mvBest})\ \mathit{xss} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{mapM\_}\ \mathit{waitCatch}\ \mathit{workers}
\end{aligned}
$$

An important feature of this function is that the failure of one worker does not affect the others, since *waitCatch* will silently ignore any worker which has thrown an exception. While deceptively succinct, these two functions provide a powerful mechanism by which multiple concurrent workers can be spawned to improve the value of a solution, all the while dealing with exceptions in a safe way by allowing the best known solution to prevail in the case of failure.

Since we can rely on the fact that the best solution will not be lost when the workers fail, we can make use of this mechanism to allow the system to demand an immediate answer at any point during the computation. This fits nicely into the framework of the contest, where programs are given a set amount of time within which to find a solution, and then given a signal which raises an exception when time is up and an answer is required. To exploit this, the function *run* is used, which spawns the workers to perform the task of finding the best solution, and provides a callback that should be executed whether the computation terminates naturally, or an exception is thrown.

```
run :: (Ord s, NFData s, NFData a) ⇒
       (a, s) → [[(a, s)]] → ((a, s) → IO ()) → IO ()
run best xss callback = catchUserInterrupt $
  bracket (newMVar best)
          (λmvBest → takeMVar mvBest ≫= callback)
          (λmvBest → spawnWorkers mvBest xss)
```

The function $bracket :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$ takes three arguments: the initial computation, which initialises the best result found so far, the final computation, which reads the best result found and calls the callback, and the intermediate computation, which spawns the workers and waits for all threads for finish. The final computation of a *bracket* is performed even if an exception is raised, which is precisely the behaviour required here when the callback is an action which outputs the best known solution.

One problem remains: if an exception is raised within a *bracket*, then after the final computation has been executed the exception will be re-raised so that it can be handled elsewhere in the system. If left unhandled, the program would exit and indicate that there was an error. The *catchUserInterrupt* function is a helper which allows the program to gracefully exit when the interrupt signal which is expected from the judging environment is received.

```
catchUserInterrupt :: IO () → IO ()
catchUserInterrupt = handle (λe → case e of UserInterrupt → return ()
                                            _             → throwIO e)
```

Note that if the exception received is not one that is expected, then the exception is thrown again and allowed to propagate further.

For testing purposes it is convenient to be able to kill worker threads after a particular amount of time, in order to simulate the judging environment. This is implemented using the *timeout* function which runs an *IO* computation within a thread and kills the thread if no result is returned within a given time limit.

```
runWithTimeout :: (Ord s, NFData s, NFData a)
                  ⇒ Int → (a, s) → [[(a, s)]] → ((a, s) → IO ()) → IO ()
runWithTimeout t best xss callback = timeout t (run best xss callback) ≫ return ()
```

This works as expected since exceptions are used to kill a thread that has expired.

6 Changing Specifications

One of the challenges was to deal with changing specifications. This was very easy to cope with in our model, and only minor extensions were required, mostly confined to the *Mine* and *Tile* datatypes, and the *stepRobot* and *updRocks* functions. On average, about 20 lines of code were added for each extension. The construction of the *GameTrie* structure relies on the *step* function to generate its branches, and so the changes in the specification are automatically reflected in the tree. As a result, all the strategies are also updated to reflect the change in specifications, since strategies use the *GameTrie* to explore possible moves.

6.1 Flooding

The first extension was to add flooding to the mines. In certain maps, there is a rising level of water. The robot operates normally underwater, but it gets destroyed if it spends too many turns underwater. Modelling flooding requires changing the *Mine* data structure, extending it to contain additional information:

```
data Mine = Mine { ...
                 , flood      :: Int
                 , waterproof :: Int
                 , water      :: Int }
```

These fields store the rate of flooding, how long the robot can last underwater, and the current level of water.

6.2 Trampolines

The second extension introduces trampolines, which act like teleporters. Once entering a trampoline, the robot gets instantly moved to a fixed destination location, and the trampoline disappears.

Similarly to flooding, trampolines requiring adding extra information to the *Mine* data structure:

```
data Mine = Mine { ...
                 , trampolines :: Set Coord
                 , targets     :: Set Coord }
```

These fields store the current position of trampolines and their associated targets. Additionally, the *stepRobot* function has to consider the case of moving into a trampoline, and we need two new tile types: trampolines and targets.

6.3 Beards and Razors

The third extension introduces beards. Beards are a new type of tile, that expand into the surrounding empty spaces in a fixed number of turns. The robot cannot traverse beards, but can collect and apply razors, which eliminate all beards surrounding the robot.

Again, the *Mine* structure has to be extended, this time with a growth factor and the number of available razors:

```
data Mine = Mine { ...
                 , growth :: Int
                 , razors :: Int }
```

Two new tile types are added (beard and razor). A new robot "movement" is to apply a razor, and the *updRocks* function now needs to update the tiles adjacent to beards as well.

6.4 Higher Order Rocks

The last extension introduces higher order rocks, which are rocks that upon impact (from falling) transform into a lambda. Each higher order rock counts as a lambda for the purpose of determining whether all lambdas have been collected.

We add a second Boolean to the *Rock* constructor to distinguish higher order rocks from normal rocks:

data *Tile* = ... | *Rock Bool Bool*

The *updRocks* function now treats higher order rocks just like ordinary rocks, apart from a small special case to check if a higher order rock should be transformed into a lambda. Additionally, the calculation of the number of lambdas after a step (*lambdas′* in Sect. 2.2) becomes more complicated. Two falling rocks can fall into the same spot, with one disappearing. If the rock that disappears is a higher order rock, then there is one fewer lambda in the mine. For simplicity, we calculate the number of remaining lambdas by traversing the entire layout:

$$lambdas' = length\ \$\ findTiles\ (\lambda t \rightarrow t \equiv Lambda \vee isRockLambda\ t)\ layout''$$

7 Conclusion

We have described our solution to the 2012 ICFP programming contest, and seen how Haskell's features are useful during fast paced prototyping. Both low-level features (such as concurrency and exception handling) and high-level features (such as purity and laziness) are key ingredients in our solution. Haskell is a mature language, with both a stable compiler and high-quality libraries. We now give some general advice for code development in similar situations, based on our experience, and reflect briefly on possible improvements to our solution.

7.1 Practical Guidelines

Testing Even though Haskell's strong type system caught many common programming errors, we still had several bugs in our code. In particular, our submitted version often returns rather poor solutions because of bugs in the simulator. We focused our development in supporting the extensions and improving the strategies, but it would have been more effective to find and eliminate bugs.

Communication Our team was split into two groups in different locations. We found that frequent short meetings were helpful to keep the team up-to-date with the whole development, while allowing individual team members to work on separate parts of the program. Video communication, and screen/application sharing is useful for distance communication, but whiteboard brainstorming is invaluable, and hard to mimic in a distance communication.

Model first We started developing our solution by writing the model (Sect. 2.1). With this in place, different team members could develop the surrounding infrastructure more or less independently. Changes to the model were discussed with everyone before being implemented, and applied as soon as possible. This helped to minimise the mismatch between different components, and to allow development in parallel effortlessly.

Pair programming We have alternated our development between whole team discussion, individual coding sessions, and pair programming. We found pair programming to be an effective way of coding the more challenging parts of our solution, with the advantage that both team members become familiar with the code.

With regard to possible improvements to our solution, while the pathfinding algorithms take care to avoid going back to the same state several times, it would be nice to have this built into the tree structure itself. However, this would mean not using a tree structure, but rather some kind of directed graph. The lazy construction of such a graph requires the use of an appropriate constructor function to be called when elements are missing in a node lookup. The details of such an implementation are beyond the scope of this paper.

We have no regrets about our choice of programming language: we found Haskell to be suitable for developing a solution to this programming contest. We had no need for features or libraries that were not available, and our solution really played to Haskell's strengths. Haskell's type system helped catch bugs early on, but we failed to test our solution against a number of simple scenarios. These bugs (all minor and easy to fix, but nonetheless present), cost us a lot of points on a number of maps, and we failed to enter the last round of the competition. In that sense, dozens of submissions outperformed ours, but our development tried to find an elegant, functional solution to the problem that was easy to adapt to changing requirements. We feel that we achieved this goal, and despite our poor final results, the sheer fun of competing in such a contest using Haskell is hard to beat.

References

1. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
2. Hinze, R.: Generalizing generalized tries. J. Funct. Program. **10**(4), 327–351 (2000)
3. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989)
4. Peyton Jones, S. (ed.): Haskell 98, Language and Libraries. The Revised Report. Cambridge University Press, Cambridge (2003). Journal of Functional Programming Special Issue 13(1)