Parallelizing a Simple Genetic Algorithm: Robbie the Robot

The purpose of my project was to implement and parallelize a simple genetic algorithm which searches for strategies for exploring a 2-d space in search of objects based on local information. For the anthropomorphization of this task I refer the reader to my proposal. I will begin with the concepts in the first code listing (Robbie.hs) which give the primitives necessary to understand the algorithm constructed in the second code listing.

The namesake of this project, Robbie, requires three data structures: a 2-d array, a place to store his position and level of achieved reward, and a genome mapping states to actions. I chose to represent these things (encapsulated in the type Sim) with nested MArray’s, a non-nested MArray, and an IntMap Label where Label is an enum which is in all cases directly mapped to the functions its enumerations represent. This is necessary for creating the practically required reverse mapping (we cannot have a Map (Sim -> IO Sim) Int without the intermediate enum because there is at least no obvious way for types Sim -> IO Sim to be Ord’ered).

The important capabilities provided by Robbie.hs for these data structures are the ability to create randomly initialized Genome’s; wrap Genome’s in fresh, randomly generated RobbieWorld’s and RobbieState’s; advance these simulations and extract scores from them; and mutate and crossover Genome’s between generations. These behaviors are represented in obvious enough ways by the exports of the Robbie module. The most important fact about its internals has to do with the choice to use arrays. Arrays in Haskell are a complicated business; I wanted to follow through on the most obvious way to make a 2-d or 1-d array without using unsafe operations as would have been necessary with the vector package—as far as I can tell. the array packages offers to forms of safety for MArray’s: through the IO monad and the ST monad. The greatest benefit of the ST monad is that it is escapable, as can be seen from the signature of the runST(U)Array functions found in Data.Array.ST. However, there is no clear point in my program where it is possible to leave arrays to be frozen and never thawed again, and Data.Array.ST offered no guidance—nor did any of the obvious places to look for Haskell instruction—on how, safely, to do this. It was not clear to me that there was in fact, a way, and escaping a monad once was not sufficient for my algorithm’s purposes. Being familiar with the IO monad (and it not requiring the trickery of an uninitialized type parameter that can only appear in function signatures), I switched to using IOArray’s and IOUArray’s. The pervasiveness of the IO monad in the Robbie module was somewhat convenient in the end, as it made random number generation possible from each of the locations that needed it (rather than splitting an extensive tree of generators at the root of the program). This choice made no difference to the sequential implementation of the algorithm in Main.hs, but it would be very consequential for the parallel version.

First let us briefly address what is implemented in Main.hs. The section labeled ”Main and Helpers” is command line argument pre-processing and a single core function which unfortunately could not in any way be parallelized: the initialization of the Genomes (the type checker did not agree with my attempt to write any combinators additional to those in Control.Monad.Par.Combinator). The core of the algorithm is in the functions evolveS and evolveP. Two small monadic combinators are very helpful to understand in reading this code, which I called iterateNM(1/2/3) (almost every permutation of the arguments was convenient at some point). iterateNM(1/2/3) simple does what Control.Monad.Loop.Iterate
does but for a finite number of steps; it concatenates the execution of \( N \) monadic actions end-to-end; it is a sure sign of the non-parallelizable components of our algorithm! Both \texttt{evolveS} and \texttt{evolveP} begin with a core loop that runs fresh simulators around each \texttt{Genome} for \texttt{nSteps} iterations, \texttt{sampleSize} times. Following some logging, rank selection is used to repopulate. As an aside, rank selection was neatly reimplemented here (with more than passing knowledge of the \texttt{random-fu} library I surely could have used a \texttt{Categorical} distribution to do the same) by selecting uniformly from an array of numbers which, modulo a certain number, represent each \texttt{genome} with the multiplicity of their rank. A mutation operation and a pairwise crossover operation are then performed on each genome and on \texttt{length genomes ‘quot’ 2} pairs of genomes, respectively. Both \texttt{evolveS} and \texttt{evolveP} return a new population of genomes (or the current one, if it is presently the last iteration) along with a mock-stateful index.

The reason that the creep of the IO monad from the previous module was worth noting is that it severely limited the parallelism option available to the \texttt{main} function. To my chagrin, it entirely ruled out the deterministic parallelism that we discussed in class. To understand why exactly, we could look at the type of the function that seems to be our closest enabler:

\[
\texttt{withStrategyIO :: Strategy a -> a -> IO a}.
\]

It would seem that given that the most important functions in our program are something like

\[
\texttt{act :: Sim -> IO Sim}
\]

or

\[
\texttt{mutateGenome :: Genome -> IO Genome}
\]

that we could make this work. Why can we not do some kind of \texttt{parMap} as offered by sublibraries of \texttt{Control.Parallel.Strategies} over the \texttt{[Sim]} or \texttt{[Genome]} types that occur at all the most expensive steps of our program? The reason is that what we really need is not just a parallel \texttt{map}, but a parallel \texttt{mapM}. While it is simple enough to \texttt{parMap} a function such as \texttt{act} to produce \texttt{[IO Sim]} or \texttt{[IO Genome]}, \texttt{withStrategyIO} or \texttt{usingIO} allow us to parallelize the reduction to normal form of the type \texttt{within} the IO monad, which is entirely sequential! We really want to parallelize the \texttt{bind} operation, which \texttt{Control.Parallel.Strategies} offers us no way to do; there is nothing that helps us with the transformation from \texttt{[IO a] -> IO [a]}. The \texttt{Par} monad as such does not help us here either; but \texttt{Control.Monad.Par.IO} does offer an IO transformer monad applied to \texttt{Par}, whose crucial capability is encapsulated in the combination of \texttt{runParIO :: ParIO a -> IO a} and the associated \texttt{MonadIO} instance providing \texttt{liftIO :: IO a -> ParIO a}. With a bidirectional translation between monads available to us, the \texttt{parMapM} operation offered by both the \texttt{Par} and \texttt{ParIO} monads can be grafted onto all of the obvious places by composing any mapping function with \texttt{liftIO} and composing the output of the map itself with \texttt{runParIO}. In our case, the non-determinism this introduces is not a problem since we are guaranteed that no dependence exists between \texttt{Genomes}; at most, there is a dependence between two \texttt{Genomes} which are being recombined, in which case the pair is the atomic unit.

Unfortunately, the parallel performance comparison was underwhelming. As a mixed Windows/Linux user I encountered insurmountable difficulties attempting to view thread-scope evaluations, so the best data I had are anecdote and the short reports emitted by the

Eli Whitehouse: \texttt{ebw2143}  
December 18, 2019
program itself such as the following on two runs with identical settings, the first with the
--par flag and the second without.

Listing 1: compiled with stack ghc -- -O2 -Wall src/Robbie.hs app/Main.hs
-threaded -rtsopts -eventlog:

bash\$ ./Main 100 10 20 5 0.5 0.001 0.8 1 log.txt --par
+RTS -N8 -s -ls -RTS
464,222,320 bytes allocated in the heap
16,158,472 bytes copied during GC
1,163,408 bytes maximum residency (6 sample(s))
122,736 bytes maximum slop
1 MB total memory in use (0 MB lost due to fragmentation)

Max pause
Gen 0  71 colls , 71 par  0.328 s 0.055 s  0.0008 s
0.0060 s
Gen 1  6 colls ,  5 par  0.078 s 0.008 s  0.0013 s
0.0030 s

Parallel GC work balance: 34.93\% (serial 0\%, perfect 100\%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 0(0 converted, 0 overflowed, 0 dud, 0 GC’d, 0 fizzled)

INIT time  0.000 s ( 0.004 s elapsed)
MUT time   7.734 s ( 8.409 s elapsed)
GC time    0.406 s ( 0.063 s elapsed)
EXIT time  0.000 s ( 0.009 s elapsed)
Total time 8.141 s ( 8.485 s elapsed)

Alloc rate  60,020,663 bytes per MUT second

Productivity 95.0\% of total user, 99.1\% of total elapsed

Listing 2: compiled with stack ghc -- -O2 -Wall src/Robbie.hs app/Main.hs
-threaded -rtsopts -eventlog:

bash\$ ./Main 100 10 20 5 0.5 0.001 0.8 1 log.txt +RTS -N8 -s -ls -RTS
371,853,112 bytes allocated in the heap
33,945,088 bytes copied during GC
1,556,792 bytes maximum residency (13 sample(s))
495,304 bytes maximum slop
1 MB total memory in use (0 MB lost due to fragmentation)
Parallel Functional Programming [COMS 4995-003]: Project Report

Max pause Tot time (elapsed) Avg pause
Gen 0 345 colls, 345 par 0.250s 0.051s 0.0001s
0.0013s
Gen 1 13 colls, 12 par 0.000s 0.012s 0.0009s
0.0025s

Parallel GC work balance: 29.00\% (serial 0\%, perfect 100\%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using −N8)

SPARKS: 0(0 converted, 0 overflowed, 0 dud, 0 GC’d, 0 fizzled)

INIT time 0.000 s ( 0.013 s elapsed)
MUT time 0.406 s ( 0.244 s elapsed)
GC time 0.250 s ( 0.063 s elapsed)
EXIT time 0.000 s ( 0.002 s elapsed)
Total time 0.656 s ( 0.322 s elapsed)

Alloc rate 915,330,737 bytes per MUT second

Productivity 61.9\% of total user, 75.9\% of total elapsed

It is thoroughly puzzling to me why the opaque facilities of the ParIO monad did not do the program any service here. It appears that however poorly-grained the load balancing may be (Control.Monad.Par.Combinators does not provide a lot of chunking faculty that I understood how to use) there simply was no forking of tasks beyond one processor, and no sparks were created even though threading is enabled and all cores were made available. Sometimes when directly using stack run, it appears that the parallelized version completes faster, though this is very hard to measure with the overhead tasks building seems to incur, but neither the messages shown above nor the eventlog appear in any place I can locate. While ultimately as far as coding style is concerned, the algorithm was parallelizable in a very modular way, it appears to have failed to generate any benefit in this case, largely due to the restrictions of such pervasive work in the IO monad, out of necessity.

Listing 3: Robbie.hs (datastructures and transformations)

```haskell
module Robbie ( Sim , Genome , mkSimWithGenome , mkGenome , act , crossGenomes , mutateGenome , readScore , ratioFromFloat ) where
```

Eli Whitehouse: ebw2143 December 18, 2019
− function imports:
−
import System.Random (randomRs, randomR,
    getStdRandom, randomRIO,
    newStdGen )
import Data.Array.MArray (readArray, writeArray ,
    newArray, newListArray )
import Data.Ratio (approxRational, numerator, denominator )
import Control.Monad (guard, forM_ )
import Data.Maybe (catMaybes, fromJust )
import Data.List (foldl' )

− type imports:
−
import Data.Array.IO (IOUArray, IOArray )
import Data.Word (Word8 )
import Data.IntMap.Strict (IntMap )
import Data.Map (Map )
import Control.DeepSeq (NFData ( . . ) )

− qualified imports:
−
import qualified Data.IntMap as IM
import qualified Data.Map as M
import qualified Data.IntSet as IS

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

{− Types, Instances, Type Synonyms −}

data Sim = Wrap RobbieWorld Genome RobbieState

data Label = MvRand
    | North
    | South
    | East
    | West
    | Stay
    | Collect
deriving (Eq, Ord, Show)

instance NFData Label where
    rnf a = a 'seq' ()

instance NFData Sim where
    rnf (Wrap rw gnm rs) = rnf gnm 'seq' rw 'seq' rs 'seq' ()

type Action = (Sim -> IO Sim)
type RobbieWorld = IOArray Int (IOUArray Int Word8)
type Genome = IntMap Label

type RobbieState = IOArray Word8 Int

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

{− Constants −}

mistakePenalty :: Int
mistakePenalty = 5

rewardCollect :: Int
rewardCollect = 1

globalEps :: (RealFrac c) => c
globalEps = 0.001

Eli Whitehouse: ebw2143  December 18, 2019
82  labels :: [Label]
83  labels = [MvRand, North, South, East, West, Stay, Collect]
84
85  actions :: [Action]
86  actions = [mvRand, north, south, east, west, stay, collect]
87
88  hashes :: [Int]
89  hashes = do
90      let vs = [0..2]
91      n <- vs
92      s <- vs
93      e <- vs
94      w <- vs
95      h <- vs
96      guard (h /= 0)
97      guard (length $ filter (==0) [n, s, e, w, h]) < 3)
98      return $ hashLoc h s n e w
99
100 actMap :: Map Label Action
101 actMap = M.fromList $ zip labels actions
102
103 labelMap :: IntMap Label
104 labelMap = IM.fromList $ zip [1..7] labels
105
106 rLabelMap :: Map Label Int
107 rLabelMap = M.fromList $ zip labels [1..7]
108
109
110 {− Evolution −}
111
112 mutateGenome :: (RealFrac c) => c -> Genome -> IO Genome
113 mutateGenome frac gnm = do
114      let size = IM.size gnm
115          (nm, dnm) = ratioFromFloat frac
116          rs <- sequence $ replicate size $ randomRIO (1, dnm)
117          g <- newStdGen
118          let keys = map snd $ filter dropRatio $ zip rs $ IM.keys gnm
119          if r > nm then False else True
120          vs = noNothingLkup rLabelMap IM.lookup $ noNothingLkup gnm IM.lookup keys
121          mutns = mapMutants vs $ randomRs (1, 7) g
122          newIMap = foldl 'mdf' gnm $ zip keys $ noNothingLkup labelMap IM.lookup mutns
123          return newIMap
124
125 mapMutants :: (Eq a) => [a] -> [a] -> [a]
126 mapMutants e@(b:bs) (c:cs) | c /= b = c : mapMutants bs cs
127                  | otherwise = mapMutants e cs
128
129 mdf :: Genome -> (Int, Label) -> Genome
130 mdf gnm (k,a) = IM.update (\_ -> Just a) k gnm
131
132 crossGenomes :: Genome -> Genome -> IO (Genome, Genome)
133 crossGenomes gnmA gnmB = do
134      let n = IM.size gnmA
135      r <- randomRIO (1, n - 1)
136      let part = IS.fromAscList $ take r $ IM.keys gnmA
137          (btmA, topA) = IM.partitionWithKey (\k -> IS.member k part) gnmA
138          (btmB, topB) = IM.partitionWithKey (\k -> IS.member k part) gnmB
139          return (btmA `IM.union` topB, btmB `IM.union` topA)
140
141 {− Stepping Simulations Forward −}
act :: Action
act sim@(Wrap rw gnm rs) = do
  sur <- sense rw rs
  step = fromJust $ M.lookup lbl actMap
  lbl = fromJust $ IM.lookup sur gnm
  step sim

sense :: RobbieWorld -> RobbieState -> IO Int
sense rw rs = do
  x <- readArray rs 1
  y <- readArray rs 2
  hashLoc <$> access2d (x, y) rw
  <$> access2d (x+1, y) rw
  <$> access2d (x-1, y) rw
  <$> access2d (x, y+1) rw
  <$> access2d (x, y-1) rw

access2d :: (Int, Int) -> RobbieWorld -> IO Word8
access2d (x, y) rw = do
  row <- readArray rw x
  readArray row y

mvRand :: Action
mvRand sim = do
  r <- getStdRandom $ randomR (2,5)
  go = fromJust $ M.lookup lbl actMap
  lbl = fromJust $ IM.lookup r labelMap
  go sim

collect :: Action
collect sim@(Wrap rw gnm rs) = do
  x <- readArray rs 1
  y <- readArray rs 2
  h <- access2d (x,y) rw
  if h /= 2 then return sim
  else do
    row <- readArray rw x
    writeArray row y 1
    score <- readArray rs 0
    writeArray rs 0 $ score + rewardCollect
    return $ Wrap rw gnm rs

stay :: Action
stay sim = return sim

north :: Action
north = move (1,0)
south :: Action
south = move (-1,0)
east :: Action
east = move (0,1)
west :: Action
west = move (0,-1)

move :: (Int, Int) -> Sim -> IO Sim
move (i,j) (Wrap rw gnm rs) = do
  let idx = if i == 0 then 2 else 1
  ent <- fetchRel (i,j) rs rw
  if ent /= 0 then do
cur <- readArray rs idx
writeArray rs idx $ cur + i + j
return $ Wrap rw gnm rs
else do
cur <- readArray rs 0
writeArray rs 0 $ cur + mistakePenalty
return $ Wrap rw gnm rs

fetchRel :: (Int, Int) -> RobbieState -> RobbieWorld -> IO Word8
fetchRel (i, j) rs rw = do
  x <- readArray rs 1
  y <- readArray rs 2
  access2d (x + i, y + j) rw

{- "Constructors" -}

mkSimWithGenome :: Int -> Float -> Genome -> IO Sim
mkSimWithGenome n frac gnm = do
  rw <- makeRW n frac
  rs <- mkRobbieState n
  return $ Wrap rw gnm rs

mkRobbieState :: Int -> IO RobbieState
mkRobbieState n = do
  rs <- sequence $ replicate 2 $ randomRIO (1, n)
  let es = 0 : (rs ++ [0])
      newListArray (0, 2) es
  return $ Wrap rw gnm rs

mkGenome :: IO Genome
mkGenome = do
  rs <- sequence $ replicate (length hashes) $ randomRIO (1, 7)
  let gnm = IM.fromList $ zip hashes lbls
      lbls = noNothingLkup labelMap IM.lookup rs
  return gnm

makeRW :: (RealFrac c) => Int -> IO RobbieWorld
makeRW n frac = do
  outer <- newArray_ (0, n+1)
  forM_ [0..n+1] $ \i -> do
    row <- newArray_ (0, n+1)
    writeArray outer i row
    if i == 0 || i == n + 1
      then do
        forM_ [0..n+1] $ \j -> writeArray row j 0
      else do
        writeArray row 0 0
        writeArray row (n+1) 0
        forM_ [1..n] $ \j -> do
          r <- shift nm <$> getStdRandom (randomR (1, dnm))
          writeArray row j r
    return outer
  where
    shift s x = if x > s then 1 else 2
    (nm, dnm) = ratioFromFloat frac

{- Utilities & Abbreviations -}

noNothingLkup :: b -> (a -> b -> Maybe c) -> [a] -> [c]
noNothingLkup m lkup ks = catMaybes $ map (flip lkup m) ks

hashLoc :: Word8 -> Word8 -> Word8 -> Word8 -> Word8 -> Int
hashLoc h n s e w = sum $ zipWith (*) integralLoc powersOf3 where
  integralLoc = map fromIntegral [h, n, s, e, w]
  powersOf3 = iterate (3*) 1

readScore :: Sim -> IO Int
readScore (Wrap _ rs) = readArray rs 0

ratioFromFloat :: (RealFrac c) => c -> (Integer, Integer)
ratioFromFloat frac = (numerator rt, denominator rt)
  where rt = approxRational frac globalEps

Eli Whitehouse: ebw2143  December 18, 2019
{\textit{Fin}}
Listing 4: Main.hs (parallel and sequential evolutionary algorithm implementations)

```haskell
module Main where

import Robbie

import System.Random(randomRIO)
import System.Environment(getProgName, getArgs)
import Data.List(partition, isPrefixOf, sortBy)
import Data.Time.Clock(getCurrentTime)
import Data.Time.LocalTime(getCurrentTimeZone, utcToLocalTime)
import Data.Array(array)
import Data.Random.runRVar
import Data.Random.Extras.choicesArray
import Data.Maybe.catMaybes
import Control.Monad.sequence
import Control.Monad.Loops(concatM)
import Control.Monad.Par.IO()
import Control.Monad.Par.IO.runParIO
import Control.Monad.Trans.liftIOL

import System.IO(Handle, IOMode(..))
import Data.Random.StdRandom(..)

import qualified Data.IntMap as IM

-- Constants

sampleSize :: Int
sampleSize = 10

-- Main & Helpers

main :: IO ()
main = do
  pn <- getProgName
  (flags, params) <- fmap (partition (isPrefixOf "-")) getArgs
  let (psize, dim, nStep, nGen, canDensity, mutRate, crossRate, logFreq, logFile) = parseParams pn params
      evolve = case filter (=="--par") flags of
        ["--par"] -> evolveP
        _ -> evolveS
  h <- openFile logFile AppendMode
  ut <- getCurrentTime
  tz <- getCurrentTimeZone
  let header = show (utcToLocalTime tz ut) ++ ": " ++
                  pn ++ ": " ++ unwords params ++ " " ++ unwords flags
  hPutStrLn h header
  gms <- initGenomes psize
  let ev = evolve dim psize logFreq canDensity mutRate crossRate h
```

Eli Whitehouse: ebw2143

December 18, 2019
_ <- iterateNM1 ev nGen ((0, nGen), gnms)

hClose h

parseParams :: String -> [String] -> ([Int, Int, Int, Float, Float, Float, Int, String]

parseParams ps dim ns ng cr If fp = test = rd

| otherwise = err pn

where rd = (read ps, read dim, read ns, read ng, read cr, read If, fp)

test = and ($ (map (>) (p, d, s, g, l) = rd

where (p, d, s, g, c, m, o, l, i) = rd

parseParams ps _ = err pn

err :: [Char] -> a

err pn = error "$ " usage: " ++ pn ++ " " ++ errString where

erString = "pop-size size dim nstep ngen can-density mutn-rate crossover-rate log-frequency " ++

"log-file -par\n" ++ " " ++

": Int Int Int Float Float Int FilePath"

initGenomes :: Int -> IO [Genome]

initGenomes = sequence . flip replicate mkGenome

{- Core Step of the Algorithm (Sequential and Parallel) plus a helper -}

evolveEv dim nstep If dens mutr cross h ((i, lastRun), gnms) = do

let initScores = replicate (length gnms) 0

scores <- iterateNM3 initScores sampleSize $ \scores -> do

sims <- mSkM (mkSimWithGenome dim dens) gnms

steppdSims <- iterateNM2 nstep ($ \scores -> mSkM act sims) sims

newScores <- mSkM readScore steppdSims

return $ zipWith (+) scores' newScores

let sortedGnms = sortBy ((\(j, k) -> compare j k) $ zip scores gnms

logMsg = show i ++ ": average , " ++ show avgScore ++ " ; "

++ "top , " ++ show topScore ++ " ; "
++ "approx. median , " ++ show medScore ++ " ."

avgScore = sum scores' . div' length sortScores

topScore = head sortScores

medScore = sortScores !! (length sortScores ' quot' 2)

sortScores = map fst sortedGns

l = length sortedGnms

if i mod If == 0 || i == lastRun

then hPutStrLn h logMsg

else return ()

selGnms <- rankSel $ zip [1, 1-1..1] $ snd $ unzip sortedGnms

mutants <- mSkM (mutateGenome mutr) selGnms

let (crssNm, crssDnm) = ratioFromFloat crossr

cross = if r <= crssNm then uncurry crossGenomes else return

crossDraw <- sequence $ replicate (length mutants ' quot' 2) $ randomRIO (1, crssDnm)

exchanged <- fmap mapUnpair $ sequence $ zipWith crossCrossDraw $ mapPair mutants

let newGnms = if even (length mutants) then exchanged else (last mutants) : exchanged

return $ if i /= lastRun

then ((i+1, lastRun), newGnms)

else ((lastRun, lastRun), map snd sortedGnms)

evolveEv dim nstep If dens mut r cross h ((i, lastRun), gnms) = do

let initScores = replicate (length gnms) 0

scores <- iterateNM3 initScores sampleSize $ \scores -> do

sims <- runParIO $ parMapM (liftIO . mkSimWithGenome dim dens) gnms

steppdSims <- runParIO $ parMapM (liftIO . iterateNM2 nstep act) sims

newScores <- runParIO $ parMapM (liftIO . readScore) steppdSims

return $ zipWith (+) scores' newScores

let sortedGnms = sortBy ((\(j, k) -> compare j k) $ zip scores gnms

logMsg = show i ++ ": average , " ++ show avgScore ++ " ; "

++ "top , " ++ show topScore ++ " ; "

++ "approx. median , " ++ show medScore ++ " ."

avgScore = sum scores' . div' length sortScores

topScore = head sortScores

medScore = sortScores !! (length sortScores ' quot' 2)

sortScores = map fst sortedGns

l = length sortedGnms

if i mod If == 0 || i == lastRun

then hPutStrLn h logMsg

else return ()

selGnms <- rankSel $ zip [1, 1-1..1] $ snd $ unzip sortedGnms

mutants <- mSkM (mutateGenome mutr) selGnms

let (crssNm, crssDnm) = ratioFromFloat crossr

cross = if r <= crssNm then uncurry crossGenomes else return

crossDraw <- sequence $ replicate (length mutants ' quot' 2) $ randomRIO (1, crssDnm)

exchanged <- fmap mapUnpair $ sequence $ zipWith crossCrossDraw $ mapPair mutants

let newGnms = if even (length mutants) then exchanged else (last mutants) : exchanged

return $ if i /= lastRun

then ((i+1, lastRun), newGnms)

else ((lastRun, lastRun), map snd sortedGnms)
++ "approx. median," ++ show medScore ++ "."

    avgScore = sum sortScores 'div' length sortScores
    topScore = head sortScores
    medScore = sortScores !! (length sortScores 'quot' 2)
    sortScores = map fst sortedGnsms
    l = length sortedGnsms
    if i 'mod' 1 == 0 || i == lastRun
      then hPutStrLn h logMsg
    else return ()
    selGnms <- rankSel $ zip [1..1..1] $ snd $ unzip sortedGnsms
    mutants <- runParIO $ parMapM (liftIO . mutateGenome mutr) selGnms
    let (crssNm, crssDnm) = ratioFromFloat crossr
        cross (r, (g1, g2)) = if r <= crssNm then crossGenomes g1 g2 else return (g1, g2)
        crossDraw <- sequence $ replicate (length mutants 'quot' 2) $ randomRIO (1, crssDnm)
        let rsWMuts = zip crossDraw (mapPair mutants)
        exchanged <- fmap mapUnpair $ runParIO $ parMapM (lift IO . cross) rsWMuts
        let newGnms = if even (length mutants) then exchanged else (last mutants) : exchanged
        return $ if i /= lastRun
                  then (((i+1)lastRun), newGnms)
                  else ((lastRun, lastRun), map snd sortedGnsms)

    rankSel :: [[(Int, a)]] -> IO [a]
    rankSel rankedIt = do
      chs <- flip runRVar StdRandom $ choicesArray (length rankedIt) opts
      return $ catMaybes $ map (flip IM.lookup m . flip mod top) chs
      where
        m = IM.fromList rankedIt
        top = 1 + head is
        nmods n = take n [ x + n | x <- [0..top..] ]
        ns = zip [1..] $ concat $ map nmods is
        opts = array (1, length ns) ns

{-- Monadic Combinators & Utilities --}

mapPair :: [a] -> [(a,a)]
mapPair (a:b:rs) = (a,b) : mapPair rs
mapPair [] = []

mapUnpair :: [(a,a)] -> [a]
mapUnpair ((a,b):rs) = a : b : mapUnpair rs
mapUnpair [] = []

iterateNM1 :: (Monad m) => (a -> m a) -> Int -> a -> m a
iterateNM1 f n = concatM $ replicate n f

iterateNM2 :: (Monad m) => Int -> (a -> m a) -> a -> m a
iterateNM2 = flip iterateNM1

iterateNM3 :: Monad m => a -> Int -> (a -> m a) -> m a
iterateNM3 a i f = iterateNM1 f i a

{-- Fin --}