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 `evolveS` and `evolveP` begin with a core loop that runs fresh simulators around each `Genome` for `nSteps` iterations, `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 `random-fu` library I surely could have used a `Categorical` distribution to do the same) by selecting uniformly from an array of numbers which, modulo a certain number, represent each `genome` with the multiplicity of their rank. A mutation operation and a pairwise crossover operation are then performed on each genome and on `length genomes ‘quot‘ 2` pairs of genomes, respectively. Both `evolveS` and `evolveP` return a now 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 `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:

```
withStrategyIO ::  Strategy a -> a -> IO a .
```

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

```
act ::  Sim -> IO Sim
```

or

```
mutateGenome ::  Genome -> IO Genome
```

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

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)

                                     Tot time (elapsed)  Avg pause
Max pause
  Gen  0        71 colls,    71 par    0.328s   0.055s     0.0008s
0.0060s
  Gen  1         6 colls,     5 par    0.078s   0.008s     0.0013s
0.0030s

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

```
                                     Tot time (elapsed)  Avg pause
Max 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    0.000s  (  0.013s elapsed)
  MUT     time    0.406s  (  0.244s elapsed)
  GC      time    0.250s  (  0.063s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time    0.656s  (  0.322s 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 `build`ing 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)

```
1  module Robbie (
2      Sim
3    , Genome
4    , mkSimWithGenome
5    , mkGenome
6    , act
7    , crossGenomes
8    , mutateGenome
9    , readScore
10   , ratioFromFloat
11 ) where
12
13 ------------------------------------------------------------
```

```
14
15 {-
16       function imports:
17 -}
18
19 import System.Random( randomRs, randomR,
20                        getStdRandom, randomRIO,
21                        newStdGen )
22 import Data.Array.MArray( readArray, writeArray,
23                           newArray_, newListArray )
24 import Data.Ratio( approxRational, numerator, denominator )
25 import Control.Monad( guard, forM_ )
26 import Data.Maybe( catMaybes, fromJust )
27 import Data.List( foldl' )
28
29 {-
30       type imports:
31 -}
32
33 import Data.Array.IO(IOUArray, IOArray)
34 import Data.Word(Word8)
35 import Data.IntMap.Strict(IntMap)
36 import Data.Map(Map)
37 import Control.DeepSeq(NFData(..))
38
39 {-
40       qualified imports:
41 -}
42
43 import qualified Data.IntMap as IM
44 import qualified Data.Map as M
45 import qualified Data.IntSet as IS
46
47 ----------------------------------------------------------------------
48 {- Types, Instances, Type Synonyms -}
49
50 data Sim = Wrap RobbieWorld Genome RobbieState
51 data Label = MvRand
52            | North
53            | South
54            | East
55            | West
56            | Stay
57            | Collect
58   deriving (Eq, Ord, Show)
59
60 instance NFData Label where
61   rnf a = a `seq` ()
62
63 instance NFData Sim where
64   rnf (Wrap rw gnm rs) = rnf gnm `seq` rw `seq` rs `seq` ()
65
66 type Action = (Sim -> IO Sim)
67 type RobbieWorld = IOArray Int (IOUArray Int Word8)
68 type Genome = IntMap Label
69 type RobbieState = IOUArray Word8 Int
70
71 ----------------------------------------------------------------------
72 {- Constants -}
73
74 mistakePenalty :: Int
75 mistakePenalty = 5
76
77 rewardCollect :: Int
78 rewardCollect = 1
79
80 globalEps :: (RealFrac c) => c
81 globalEps = 0.001
```

```
82
83  labels :: [Label]
84  labels = [MvRand, North, South, East, West, Stay, Collect]
85
86  actions :: [Action]
87  actions = [mvRand, north, south, east, west, stay, collect]
88
89  hashes :: [Int]
90  hashes = do
91    let vs = [0..2]
92    n <- vs
93    s <- vs
94    e <- vs
95    w <- vs
96    h <- vs
97    guard (h /= 0)
98    guard ((length $ filter (==0) [n, s, e, w, h]) < 3)
99    return $ hashLoc h s n e w
100
101 actMap :: Map Label Action
102 actMap = M.fromList $ zip labels actions
103
104 labelMap :: IntMap Label
105 labelMap = IM.fromList $ zip [1..7] labels
106
107 rLabelMap :: Map Label Int
108 rLabelMap = M.fromList $ zip labels [1..7]
109
110 ------------------------------------------------------------------------
111 {- Evolution -}
112
113 mutateGenome :: (RealFrac c) => c -> Genome -> IO Genome
114 mutateGenome frac gnm = do
115     let size = IM.size gnm
116         (nm, dnm) = ratioFromFloat frac
117
118     rs <- sequence $ replicate size $ randomRIO (1,dnm)
119     g <- newStdGen
120
121     let keys = map snd $ filter dropRatio $ zip rs $ IM.keys gnm
122         dropRatio (r,_) = if r > nm then False else True
123         vs = noNothingLkup rLabelMap M.lookup $ noNothingLkup gnm IM.lookup keys
124         mutns = mapMutants vs $ randomRs (1,7) g
125         newIMap = foldl' mdf gnm $ zip keys $ noNothingLkup labelMap IM.lookup mutns
126
127     return newIMap
128
129 mapMutants :: (Eq a) => [a] -> [a] -> [a]
130 mapMutants e@(b:bs) (c:cs) | c /= b    = c : mapMutants bs cs
131                            | otherwise = mapMutants e cs
132 mapMutants [] _ = []
133 mapMutants _ [] = []
134
135 mdf :: Genome -> (Int, Label) -> Genome
136 mdf gnm (k,a) = IM.update (\_ -> Just a) k gnm
137
138 crossGenomes :: Genome -> Genome -> IO (Genome, Genome)
139 crossGenomes gnmA gnmB = do
140     let n = IM.size gnmA
141     r <- randomRIO (1, n - 1)
142     let part = IS.fromAscList $ take r $ IM.keys gnmA
143         (btmA, topA) = IM.partitionWithKey (\k _ -> IS.member k part) gnmA
144         (btmB, topB) = IM.partitionWithKey (\k _ -> IS.member k part) gnmB
145     return (btmA `IM.union` topB, btmB `IM.union` topA)
146
147 ------------------------------------------------------------------------
148 {- Stepping Simulations Forward -}
149
```

```
150 act :: Action
151 act sim@(Wrap rw gnm rs) = do
152     sur <- sense rw rs
153     let step = fromJust $ M.lookup lbl actMap
154         lbl = fromJust $ IM.lookup sur gnm
155     step sim
156
157 sense :: RobbieWorld -> RobbieState -> IO Int
158 sense rw rs = do
159     x <- readArray rs 1
160     y <- readArray rs 2
161     hashLoc <$> access2d (x,y) rw
162         <*> access2d (x+1, y) rw
163         <*> access2d (x-1, y) rw
164         <*> access2d (x, y+1) rw
165         <*> access2d (x, y-1) rw
166
167 access2d :: (Int, Int) -> RobbieWorld -> IO Word8
168 access2d (x, y) rw = do
169     row <- readArray rw x
170     readArray row y
171
172 mvRand :: Action
173 mvRand sim = do
174     r <- getStdRandom $ randomR (2,5)
175     let go = fromJust $ M.lookup lbl actMap
176         lbl = fromJust $ IM.lookup r labelMap
177     go sim
178
179 collect :: Action
180 collect sim@(Wrap rw gnm rs) = do
181     x <- readArray rs 1
182     y <- readArray rs 2
183     h <- access2d (x,y) rw
184     if h /= 2
185       then return sim
186       else do
187         row <- readArray rw x
188         writeArray row y 1
189         score <- readArray rs 0
190         writeArray rs 0 $ score + rewardCollect
191         return $ Wrap rw gnm rs
192
193 stay :: Action
194 stay sim = return sim
195
196 north :: Action
197 north = move (1,0)
198
199 south :: Action
200 south = move (-1,0)
201
202 east :: Action
203 east = move (0,1)
204
205 west :: Action
206 west = move (0,-1)
207
208 move :: (Int, Int) -> Sim -> IO Sim
209 move (i,j) (Wrap rw gnm rs) = do
210     let idx = if i == 0 then 2 else 1
211     ent <- fetchRel (i,j) rs rw
212     if ent /= 0
213       then do cur <- readArray rs idx
214               writeArray rs idx $ cur + i + j
215               return $ Wrap rw gnm rs
216       else do cur <- readArray rs 0
217               writeArray rs 0 $ cur - mistakePenalty
```

```
218                 return $ Wrap rw gnm rs
219
220 fetchRel :: (Int, Int) -> RobbieState -> RobbieWorld -> IO Word8
221 fetchRel (i, j) rs rw = do
222     x <- readArray rs 1
223     y <- readArray rs 2
224     access2d (x + i, y + j) rw
225
226 ------------------------------------------------------------------------
227 {- "Constructors" -}
228
229 mkSimWithGenome :: Int -> Float -> Genome -> IO Sim
230 mkSimWithGenome n frac gnm = do
231   rw <- makeRW n frac
232   rs <- mkRobbieState n
233   return $ Wrap rw gnm rs
234
235 mkRobbieState :: Int -> IO RobbieState
236 mkRobbieState n = do
237     rs <- sequence $ replicate 2 $ randomRIO (1,n)
238     let es = 0 : (rs ++ [0])
239     newListArray (0,2) es
240
241 mkGenome :: IO Genome
242 mkGenome = do
243   rs <- sequence $ replicate (length hashes) $ randomRIO (1,7)
244   let gnm = IM.fromList $ zip hashes lbls
245       lbls = noNothingLkup labelMap IM.lookup rs
246   return gnm
247
248 makeRW :: (RealFrac c) => Int -> c -> IO RobbieWorld
249 makeRW n frac = do
250     outer <- newArray_ (0,n+1)
251     forM_ [0..n+1] $ \i -> do
252       row <- newArray_ (0,n+1)
253       writeArray outer i row
254       if i == 0 || i == n + 1
255         then do
256             forM_ [0..n+1] $ \j -> writeArray row j 0
257         else do
258             writeArray row 0 0
259             writeArray row (n+1) 0
260             forM_ [1..n] $ \j -> do
261               r <- shift nm <$> getStdRandom (randomR (1,dnm))
262               writeArray row j r
263     return outer
264   where
265     shift s x = if x > s then 1 else 2
266     (nm, dnm) = ratioFromFloat frac
267
268 ------------------------------------------------------------------------
269 {- Utilities & Abbreviations -}
270
271 noNothingLkup :: b -> (a -> b -> Maybe c) -> [a] -> [c]
272 noNothingLkup m lkup ks = catMaybes $ map (flip lkup m) ks
273
274 hashLoc :: Word8 -> Word8 -> Word8 -> Word8 -> Word8 -> Int
275 hashLoc h n s e w = sum $ zipWith (*) integralLoc powersOf3 where
276     integralLoc = map fromIntegral [h,n,s,e,w]
277     powersOf3 = iterate (3*) 1
278
279 readScore :: Sim -> IO Int
280 readScore (Wrap _ _ rs) = readArray rs 0
281
282 ratioFromFloat :: (RealFrac c) => c -> (Integer, Integer)
283 ratioFromFloat frac = (numerator rt, denominator rt)
284   where rt = approxRational frac globalEps
285
```

286 ——————————————————————————————————————
287 ——————————————————————————————————————
288 $\{-$ Fin $-\}$
289 ——————————————————————————————————————
290 ——————————————————————————————————————

Listing 4: Main.hs (parallel and sequential evolutionary algorithm implementations)

```
1  module Main where
2
3  ------------------------------------------------------------------------------
4
5  import Robbie
6
7  {-
8        function imports:
9  -}
10
11 import System.Random( randomRIO )
12 import System.Environment( getProgName, getArgs )
13 import System.IO( openFile, hClose, hPutStrLn )
14 import Data.List( partition, isPrefixOf, sortBy )
15 import Data.Time.Clock( getCurrentTime )
16 import Data.Time.LocalTime( getCurrentTimeZone, utcToLocalTime )
17 import Data.Array( array )
18 import Data.Random( runRVar )
19 import Data.Random.Extras( choicesArray )
20 import Data.Maybe( catMaybes )
21 import Control.Monad( sequence )
22 import Control.Monad.Loops( concatM )
23 import Control.Monad.Par.IO()
24 import Control.Monad.Par.IO( runParIO )
25 import Control.Monad.Par.Combinator( parMapM )
26 import Control.Monad.Trans( liftIO )
27
28 {-
29       type imports:
30 -}
31
32 import System.IO(Handle, IOMode(..))
33 import Data.Random(StdRandom(..))
34
35 {-
36       qualified imports:
37 -}
38
39 import qualified Data.IntMap as IM
40
41 ------------------------------------------------------------------------------
42 {- Constants -}
43
44 sampleSize :: Int
45 sampleSize = 10
46
47 ------------------------------------------------------------------------------
48 {- Main & Helpers -}
49
50 main :: IO ()
51 main = do
52     pn <- getProgName
53     (flags, params) <- fmap (partition (isPrefixOf "-")) getArgs
54     let (psize, dim, nStep, nGen, canDensity,
55               mutRate, crossRate, logFreq, logFile) = parseParams pn params
56         evolve = case filter (=="--par") flags of
57                     ["--par"] -> evolveP
58                     _         -> evolveS
59     h <- openFile logFile AppendMode
60     ut <- getCurrentTime
61     tz <- getCurrentTimeZone
62     let header = show (utcToLocalTime tz ut) ++ ": " ++
63                     pn ++ " " ++ unwords params ++ " " ++ unwords flags
64     hPutStrLn h header
65     gnms <- initGenomes psize
66     let ev = evolve dim nStep logFreq canDensity mutRate crossRate h
```

```haskell
    _ <- iterateNM1 ev nGen ((0, nGen), gnms)
    hClose h

parseParams :: String -> [String] -> (Int,Int,Int,Int,Float,Float,Float,Int,String)
parseParams pn (ps:dim:ns:ng:cd:mr:cr:lf:fp:_) | test       = rd
                                               | otherwise = err pn
    where rd :: (Int,Int,Int,Int,Float,Float,Float,Int,String)
          rd = (read ps, read dim, read ns, read ng, read cd, read mr, read cr, read lf, fp)
          test = and $ (map (>0) [p,d,s,g,l]) ++ (map (\x -> x > 0 && x <= 1) [c,m,o])
            where (p,d,s,g,c,m,o,l,_) = rd
parseParams pn _ = err pn

err :: [Char] -> a
err pn = error $ "usage: " ++ pn ++ " " ++ errString where
    errString = "pop-size dim nstep ngen can-density mutn-rate crossover-rate log-frequency " ++
                "log-file --par\n" ++ "                    " ++
                ":: Int Int Int Int Float Float Float Int FilePath"

initGenomes :: Int -> IO [Genome]
initGenomes = sequence . flip replicate mkGenome

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

evolveS :: Int -> Int -> Int -> Float -> Float -> Float -> Handle
                -> ((Int, Int), [Genome]) -> IO ((Int, Int), [Genome])
evolveS dim nstep lf dens mutr crossr h ((i, lastRun), gnms) = do
    let initScores = replicate (length gnms) 0
    scores <- iterateNM3 initScores sampleSize $ \scores' -> do
        sims <- mapM (mkSimWithGenome dim dens) gnms
        steppdSims <- iterateNM2 nstep (\sms -> mapM act sms) sims
        newScores <- mapM readScore steppdSims
        return $ zipWith (+) scores' newScores
    let sortedGnms = sortBy (\(j,_) (k,_) -> compare k j) $ zip scores gnms
        logMsg = show i ++ ": average, " ++ show avgScore ++ "; "
                    ++ "top, " ++ show topScore ++ "; "
                    ++ "approx. median, " ++ show medScore ++ "."
        avgScore = sum sortScores `div` length sortScores
        topScore = head sortScores
        medScore = sortScores !! (length sortScores `quot` 2)
        sortScores = map fst sortedGnms
        l = length sortedGnms
    if i `mod` lf == 0 || i == lastRun
        then hPutStrLn h logMsg
        else return ()
    selGnms <- rankSel $ zip [l,l-1..1] $ snd $ unzip sortedGnms
    mutants <- mapM (mutateGenome mutr) selGnms
    let (crssNm, crssDnm) = ratioFromFloat crossr
        cross r = if r <= crssNm then uncurry crossGenomes else return
    crossDraw <- sequence $ replicate (length mutants `quot` 2) $ randomRIO (1,crssDnm)
    exchanged <- fmap mapUnpair $ sequence $ zipWith cross crossDraw $ 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)

evolveP :: Int -> Int -> Int -> Float -> Float -> Float -> Handle
                -> ((Int, Int), [Genome]) -> IO ((Int, Int), [Genome])
evolveP dim nstep lf dens mutr crossr 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 k j) $ zip scores gnms
        logMsg = show i ++ ": average, " ++ show avgScore ++ "; "
                        ++ "top, " ++ show topScore ++ "; "
```

```
135                         ++ "approx. median, " ++ show medScore ++ "."
136         avgScore = sum sortScores `div` length sortScores
137         topScore = head sortScores
138         medScore = sortScores !! (length sortScores `quot` 2)
139         sortScores = map fst sortedGnms
140         l = length sortedGnms
141     if i `mod` lf == 0 || i == lastRun
142         then hPutStrLn h logMsg
143         else return ()
144     selGnms <- rankSel $ zip [l,l-1..1] $ snd $ unzip sortedGnms
145     mutants <- runParIO $ parMapM (liftIO . mutateGenome mutr) selGnms
146     let (crssNm, crssDnm) = ratioFromFloat crossr
147         cross (r, (g1,g2)) = if r <= crssNm then crossGenomes g1 g2 else return (g1,g2)
148     crossDraw <- sequence $ replicate (length mutants `quot` 2) $ randomRIO (1,crssDnm)
149     let rsWMuts = zip crossDraw (mapPair mutants)
150     exchanged <- fmap mapUnpair $ runParIO $ parMapM (liftIO . cross) rsWMuts
151     let newGnms = if even (length mutants) then exchanged else (last mutants) : exchanged
152     return $ if i /= lastRun
153                 then ((i+1,lastRun), newGnms)
154                 else ((lastRun,lastRun), map snd sortedGnms)
155
156 rankSel :: [(Int, a)] -> IO [a]
157 rankSel rankedIt = do
158     chs <- flip runRVar StdRandom $ choicesArray (length rankedIt) opts
159     return $ catMaybes $ map (flip IM.lookup m . flip mod top) chs
160   where
161     m = IM.fromList rankedIt
162     is = map fst rankedIt
163     top = 1 + head is
164     nmods n = take n [ x + n | x <- [0,top..] ]
165     ns = zip [1..] $ concat $ map nmods is
166     opts = array (1, length ns) ns
167
168 ------------------------------------------------------------------------
169 {- Monadic Combinators & Utilities -}
170
171 mapPair :: [a] -> [(a,a)]
172 mapPair (a:b:rs) = (a,b) : mapPair rs
173 mapPair _ = []
174
175 mapUnpair :: [(a,a)] -> [a]
176 mapUnpair ((a,b):rs) = a : b : mapUnpair rs
177 mapUnpair [] = []
178
179 iterateNM1 :: (Monad m) => (a -> m a) -> Int -> a -> m a
180 iterateNM1 f n = concatM $ replicate n f
181
182 iterateNM2 :: (Monad m) => Int -> (a -> m a) -> a -> m a
183 iterateNM2 = flip iterateNM1
184
185 iterateNM3 :: Monad m => a -> Int -> (a -> m a) -> m a
186 iterateNM3 a i f = iterateNM1 f i a
187
188 ------------------------------------------------------------------------
189 ------------------------------------------------------------------------
190                         {-      Fin      -}
191 ------------------------------------------------------------------------
192 ------------------------------------------------------------------------
```