Sketch for simulating chemodiversity

Stefan Dresselhaus

April 23, 2018

(rough) sketch components responsible for chemodiversity

Genes

- define which enzymes are produced in which quantities
  - list in fig. 1 in [1]
- can be scaled down/inactivated (i.e. when predators leave for generations)
  - easy to ramp up production as long as the genes are still there
- plants can survive without problems with inactive PSM-cycles when no adversaries are present.

Inheritance & Mutation

- via whole-genome and local-genome duplication
- copies accumulate mutations that lead to neofunctionalization
- e.g. subtle differences in terpene synthases can yield vastly different products
  - i.e. these changes can appear easily
  - need to classify products by "chemical distance" for simulation
  - **TODO**: Map/Markov-Chain of mutations that may occur here?

Evolutional strategies

- "Bet-hedging": reduce variations of fitness over time

    - **TODO**: understand
    - different effects of intra-cohort-variation vs. inter-cohort-variation
- Plants with inactive PSM can survive if predators are deterred by other individuals due to automimicry-effect which *could* foster wider genetic variance
    - the more of those individuals are present in a population, the less their overall fitness becomes.
    - **TODO**: fitness must also be able to depend on relative appearance of adversarial traits in the population
        * Keyword: Frequency-dependent-selection (FDS)

Pathways to produce chemical compounds

- 40k+ compounds just stem from compounds of the calvin-cycle taking the MEP-pathway or from the krebs-cycle taking the MVA-pathway
    - both yield the same intermediate product that forms the basis.
- 10k+ compounds are amino-acid-derivatives
- Chapter VI in [1] exemplary describes 4 complete different pathways that yield compounds.
    - similar compounds/pathways should be found in the simulation

---

Consequences of producing compounds

- taking away parts of the calvin/krebs cycle puts pressure on those
    - **TODO**: find out what they do and on what they depend.
- **TODO**: where do amino-acids come from? How much impact has the diversion of these components?

Maintaining chemical diversity

+ screening hypothesis

- many PSM found have no *known* biological activity
- plants "keep them around" in case another mutation needs them to produce something "useful"
- creating things without use increase the need for photosynthesis and/or nutrient uptake.

- screening hypothesis

- it is suggested that local abiotic & biotic selection pressures are the main driver
- inactive molecules are not maintained long
- it was observed that some plants "rediscovered" some compounds in their evolution suggesting they got rid of them when no pressure to maintain them was applied

---

questions resulting from this that should be answered in the simulation

- details in chapter VIII of [1]
- how quick can lost diversity be restored?
- how expensive is it to keep producing many inactive substances while also producing active deterrents? Does this lead to a single point-of-failure due to overspecialisation? What must be done to prevent this?
- strong selection pressure *should* decrease quantity of compounds due to costs, but plants do not seem to care.
  - is this diversity needed in presence of multiple different adversaries?
  - does the simulation specialize when only presented with one adversary? What about adaptive adversaries?
  - adaptation in the qualitative & quantitative evolution in response to changed pressure? (i.e. those who cannot adapt quick enough die?)

Scenario

As this is literate Haskell first a bit of throat-clearing:

```haskell
{-# LANGUAGE RecordWildCards #-}

import Data.Functor         ((<$>))
import Control.Applicative ((<*>))
import Control.Monad        (forM_)
import Data.List            (permutations, subsequences)
```

Then some general aliases to make everything more readable:

```haskell
type Probability = Float
type Quantity = Int
type Activation = Float
type Amount = Float
```

Nutrients & Compounds

Nutrients are the basis for any reaction and are found in the environment of the plant.

```haskell
data Nutrient = Sulfur
              | Phosphor
              | Nitrate
              | Photosynthesis
     deriving (Show, Enum, Bounded, Eq)

data Component = PP
               | FPP
     deriving (Show, Enum, Bounded, Eq)
```

Compounds are either direct nutrients or already processed components

```haskell
data Compound = Substrate Nutrient
              | Produced Component
     deriving (Show, Eq)
```

This simple definition is only a brief sketch.

Enzymes

Enzymes are the main reaction-driver behind synthesis of intricate compounds.

```haskell
data Enzyme = Enzyme
       { enzymeName            :: String
         -- ^ Name of the Enzyme. Enzymes with the same name are supposed
         --   to be identical.
       , substrateRequirements :: [(Nutrient,Amount)]
         -- ^ needed for reaction to take place
       , substrateIntolerance  :: [(Nutrient,Amount)]
         -- ^ inhibits reaction if given nutrients are above the given concentration
       , synthesis             :: [(Compound,Amount)] -> [(Compound,Amount)]
         -- ^ given x in amount a, this will produce y in amount b
       , dominance             :: Maybe Amount
         -- ^ in case of competition for nutrients this denotes the priority
         --   Nothing = max possible
       }

instance Show Enzyme where
  show (Enzyme{..}) = enzymeName

instance Eq Enzyme where
  a == b = enzymeName a == enzymeName b
```

Example “enzymes” could be:

```haskell
pps :: Enzyme -- uses Phosphor from Substrate to produce PP
pps = Enzyme "PPS" [(Phosphor,1)] [] syn Nothing
```

```
 where
   syn compAvailable = [(Substrate Phosphor,i*(-1)),(Produced PP,i)]
     where
       i = getAmountOf (Substrate Phosphor) compAvailable

fpps :: Enzyme -- PP -> FPP
fpps = Enzyme "FPPS" [] [] syn Nothing
 where
   syn compAvailable = [(Produced PP,i*(-1)),(Produced FPP,i*0.5)]
     where
       i = getAmountOf (Produced PP) compAvailable
```

Environment

In the environment we have predators that impact the fitness of our plants and may be resistant to some compounds the plant produces. They can also differ in their intensity.

```
data Predator = Predator { resistance    :: [Component]
                           -- ^ list of components this predator is resistant to
                         , fitnessImpact :: Amount
                           -- ^ impact on the fitness of a plant
                           --   (~ agressiveness of the herbivore)
                         } deriving (Show, Eq)
```

Exemplatory:

```
greenfly :: Predator         -- 20% of plants die to greenfly, but the fly is
greenfly = Predator [PP] 0.2 -- killed by any Component not being PP
```

The environment itself is just the soil and the predators. Extensions would be possible.

```
data Environment =
     Environment
   { soil      :: [(Nutrient, Amount)]
     -- ^ soil is a list of nutrients available to the plant.
   , predators :: [(Predator, Probability)]
     -- ^ Predators with the probability of appearance in this generation.
   } deriving (Show, Eq)
```

Example:

```
exampleEnvironment :: Environment
exampleEnvironment =
  Environment
    { soil = [ (Nitrate, 2)
             , (Phosphor, 3)
             , (Photosynthesis, 10)
             ]
    , predators = [ (greenfly, 0.1) ]
    }
```

Plants

Plants consist of a Genome responsible for creation of the PSM and also an internal state how many nutrients and compounds are currently inside the plant.

```
type Genome = [(Enzyme, Quantity, Activation)]

data Plant = Plant
     { genome             :: Genome
       -- ^ the genetic characteristic of the plant
     , absorbNutrients    :: Environment -> [(Nutrient,Amount)]
       -- ^ the capability to absorb nutrients given an environment
     }
instance Show Plant where
  show p = "Plant with Genome " ++ show (genome p)
instance Eq Plant where
  a == b = genome a == genome b
```

The following example yields in the example-environment this population:

```
*Main> printPopulation [pps, fpps] plants
Population:
PPS     ______oöö+++______oöö+++____________oöö+++oöö+++
FPPS    ____________oöö+++oöö+++______oöö+++______oöö+++
```

```
plants :: [Plant]
plants = (\g -> Plant g defaultAbsorption) <$> genomes
 where
   enzymes = [pps, fpps]
   quantity = [1,2] :: [Quantity]
   activation = [0.7, 0.9, 1]

   genomes = do
     e <- permutations enzymes
     e' <- subsequences e
     q <- quantity
     a <- activation
     return $ (,,) <$> e' <*> [q] <*> [a]

   defaultAbsorption (Environment s _) = limit Phosphor 2
                                       . limit Nitrate 1
                                       . limit Sulfur 0
                                       <$> s
   -- custom absorbtion with helper-function:
   limit :: Nutrient -> Amount -> (Nutrient, Amount) -> (Nutrient, Amount)
   limit n a (n', a')
     | n == n'   = (n, min a a') -- if we should limit, then we do ;)
     | otherwise = (n', a')
```

Fitness

The fitness-measure is central for the generation of offspring and the simulation. It evaluates the probability for passing on genes given a plant in an environment.

```
type Fitness = Float

fitness :: Environment -> Plant -> Fitness
fitness e p = survivalRate
   where
     nutrients = absorbNutrients p e
     products = produceCompounds p nutrients
     survivalRate = deterPredators (predators e) products
```

```
produceCompounds :: Plant -> [(Nutrient, Amount)] -> [Compound]
produceCompounds (Plant genes _) = undefined
  -- this will take some constrained linear algebra-solving

deterPredators :: [(Predator, Probability)] -> [Compound] -> Probability
deterPredators ps cs = sum $ do
      c <- cs -- for every compound
      (p,prob) <- ps -- and every predator
      return (if c `notin` (resistance p) -- if the plant cannot deter the predator
                 then prob * fitnessImpact p -- impact it weighted by probability
                 else 0)
   where
     (Produced a) `notin` b = all (/=a) b
     _ `notin`_             = False
```

Mating & Creation of diversity

TODO

Running the simulation

```
main = do
  putStrLn "Environment:"
  print exampleEnvironment
  putStrLn "Example population:"
  printPopulation [pps, fpps] plants
```

```
runhaskell sketch.md.lhs
Environment:
Environment { soil = [(Nitrate,2.0),(Phosphor,3.0),(Photosynthesis,10.0)]
            , predators = [(Predator {resistance = [PP], fitnessImpact = 0.2},0.1)]}
Example population:
Population:
```

```
PPS      ______oöö+++______oöö+++____________oöö+++oöö+++
FPPS     ____________oöö+++oöö+++______oöö+++______oöö+++
```

Utility Functions

```haskell
getAmountOf :: Compound -> [(Compound, Amount)] -> Amount
getAmountOf c = sum . fmap snd . filter ((== c) . fst)

printPopulation :: [Enzyme] -> [Plant] -> IO ()
printPopulation es ps = do
  let padded i str = take i $ str ++ repeat ' '
  putStrLn "Population:"
  forM_ es $ \e -> do
    putStr $ padded 8 (show e)
    forM_ ps $ \(Plant g _) -> do
      let curE = sum $ map (\(_,q,a) -> (fromIntegral q)*a)
                     . filter (\(e',_,_) -> e == e')
                     $ g
          plot x
           | x > 2     = "O"
           | x > 1     = "+"
           | x > 0.7   = "ö"
           | x > 0.5   = "o"
           | x > 0     = "."
           | otherwise = "_"
      putStr (plot curE)
    putStrLn ""
```