Genetic algorithms in Haskell with polytypic programming

Genetiska algoritmer i Haskell
med polytypisk programmering

Måns Vestin
email: md3msns@mdstud.chalmers.se

February 27, 1997

Abstract

Evolution is the main optimising tool of the nature. You just have to look at all the animals, plants and human beings to be amazed by their complex and well suited genes. This is not the work of a random search through all possible genotypes, it is the result of a general and efficient optimising algorithm.

Genetic algorithms and genetic programming are methods based on the concept of evolution and Darwinian natural selection. They are powerful methods of finding solutions to a vide variety of problems. This text describes a program written in the functional language Haskell, where evolution is used to solve three example problems. Two examples concern picture generation, and the third is a planning problem. Due to the use of polytypism, the program is easily extended to solve new problems using the same algorithm.

1 Genetic algorithms

Assume that we want to solve a given problem P. The set of possible solutions is D. Assume also that there is a way of telling how good a specific solution is, a way of determining its *fitness value*.

To find a good solution, we start by choosing a random set of solutions from D, the first generation of solutions. In every step of the genetic algorithm, the fitness values of the individuals of the current generation are evaluated. These values represent the probability of surviving until the next generation. The surviving solutions are combined using genetic crossover and/or random mutations, and the surviving genes and their children will together create a new generation of solutions. This new generation will hopefully have a higher fitness average than the previous. When the fitness values satisfy some predefined criteria, or when they seem to have converged, the best solution of the last generation is chosen as the output of the genetic algorithm.

Genetic programming is a special case of the algorithm described above, were the solution space D consists of programs trying to solve a problem. The genes will be computer programs, which are run and evaluated by comparing their output with some desired result.

- example 1: P is "Find a program which draws a good looking picture". The elements of D should describe functions taking x and y coordinates and returning a colour or light intensity. D might be:

  ```
  Expr = Num Real             // Constant numbers
       | Xcoord               // The coordinates of the point
       | Ycoord
       | BinOp {+,-,*,/} Expr Expr
                              // A function and two expressions
  ```

  This expression is evaluated for every point in the image, giving values which are mapped to colours. A simple gene could be `BinOp + XCoord Ycoord`, which would give a picture with low values (dark colours) in the bottom left corner and high values (bright colours) in the top right.

- example 2: P is "Write a program which guides an ant through a labyrinth, finding as much food as possible". The labyrinth can be a square grid with food units at some of the points. An ant is described by a plan of type

  ```
  Plan = If-food-ahead Plan Plan
       | Move
       | TurnLeft
       | TurnRight
       | Seq Plan Plan
  ```

  A plan might be `If-food-ahead Move (Seq TurnLeft Move)`, which would make an ant go forward if there is food in front of it, or else turn to the left and move in that direction.

- example 3: P is "Find a program which generates insect-like line drawings". D is the set of lists of nine integer numbers in some given interval. These numbers are used as angles by a recursive tree-drawing function. For a similar example, see [9].

We begin by generating a first generation of solutions, in our examples random expressions or plans. In every step of the algorithm each individual in the generation is evaluated. The fitness values are then used to choose the best individuals. These well fitted solutions are combined and a new generation is created.

- Since we can't define a function which tells us if a picture is good looking, the fitness values have to be provided by the user. To combine two functions x and y, we can exchange a sub-expression in x by a sub-expression from y.
- In the second example, the fitness of an ant is the number of found food items. The plan is executed over and over again, for a limited number of steps. A step is one `Move`, `TurnLeft` or `TurnRight` operation. Crossover can be defined as in the previous example.
- As in the first example, the user must provide the fitness values of these drawings. Two genes are combined by taking n elements from the first list and (9-n) elements from the second, resulting in a new list of nine numbers.

Instead of using only crossover, we can apply random mutation to the genes to expand the gene pool. If the solutions seem to be stuck in some kind of local minima, it might be a good idea to use mutations instead of crossover, or both.

More information about genetic programming can be found in [8].

1.1 An extended example, planning

We will now look closer at the planning example. We use a 64*64 grid, with a trace of food units as shown in the picture. To evaluate the fitness of an ant, we place it in the upper left corner facing right and evaluate its plan repeatedly until 250 time steps has past.

Since there are 89 food units along the track, the maximum fitness value is 89. The populations consist of 25 genes, and they are of the datatype

```
datatype Plan = If-food-ahead Plan Plan
              | Move
              | TurnLeft
              | TurnRight
              | Seq Plan Plan
```

In the first random generation, the best ant found 11 food units, that is all food until the first gap in the trace. If we examine the ant's gene, it is the plan

```
Seq
  (Seq
```

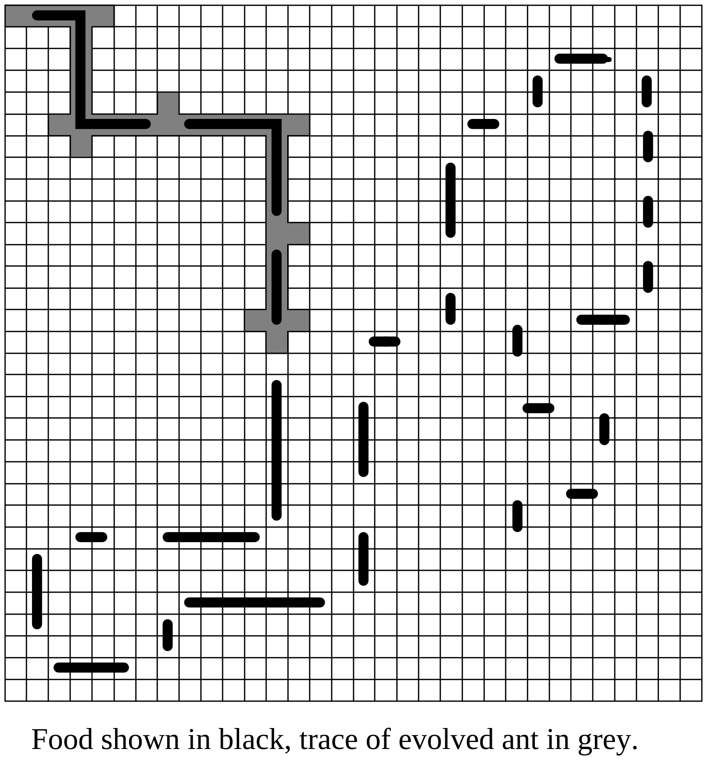

Figure 1: Food is shown in black, trace of evolved ant in grey.

```
    (If-food-ahead
      (If-food-ahead
        Move
        (Seq TurnLeft TurnLeft))
      TurnRight)
    TurnLeft)
  TurnLeft
```

which can be reduced to:

```
If-food-ahead
  (Seq Move (Seq TurnLeft TurnLeft))
  TurnLeft
```

If there is food in front of the ant, it will move one step forward and eat it, and then turn around. If not, it spins counterclockwise until food is seen.

Other ants in the first example are not as complex, for example there are several individuals with plans consisting of just a `TurnLeft` or `TurnRight`. They will not find any food, since they are not moving. There are also the `Move` ants, who run straight forward and find three food items.

For each individual in the next generation, one or two parents are chosen, depending on if we are using mutation or crossover. If an ant has a high fitness value, the chance that this ant is used as a parent is also high. In the crossover case, one sub-plan in the first parent is exchanged for a sub-plan from the other parent. In the mutation case, a sub-plan is replaced with a new random plan.

The average number of found food units are 1.72 after the first generation. After the second, it has raised to 2.32, and the maximum fitness is 15. But this best ant does not follow the trace, it just runs through the maze in a straight, diagonal line. This means that it is more vulnerable to mutations than wiser ants, because it has no strategy that can be inherited by its children. Any crossover or mutation operation will probably change its behaviour and hence destroy its high fitness value. It is gone already in the next generation. In the following generations, the maximum fitness remains at the first value of 11, but the average increases as more and more ants find out about the first 11 food units. But in generation 15 something happens. One ant discovers a way of moving on when it encounters a gap in the trace. It follows the first continuous units and then runs straight down until it finds more food. Then it follows this new trace. It finds 20 units of food before the time is up, and is the best ant ever.

The same pattern now repeats. The maximum fitness stays at 20 but the average increases towards this limit, until suddenly a mutation or crossover operation makes one of the ants smarter. This new ant has the rather complex plan

```
If-food-ahead (If-food-ahead Move (Seq (If-food-ahead Move Move)
(Seq TurnLeft Move))) (Seq Move (Seq (If-food-ahead TurnRight
(Seq TurnLeft (Seq (Seq TurnLeft (If-food-ahead Move
(If-food-ahead Move Move))) (Seq TurnLeft TurnLeft)))) TurnRight))
```

which can be reduced to:

```
If-food-ahead
  Move
  (Seq
    Move
    (Seq
      (If-food-ahead
        TurnRight
        (Seq TurnLeft (Seq (TurnLeft Move)))
      Right))
```

It makes its way through the first two single unit gaps but stops at the third gap, where two food units are missing in the trail. Its fitness reaches 24. The path of this ant is shown in grey in the picture above.

This pattern is repeated, many slow increases in average fitness are altered with sudden jumps in the maximum fitness. As the fitness values get higher, the number of generations between each jump increases. The size of the plans also increases with time, and unfortunately the program tends to run out of memory before any really good ant evolves.

In figure 2, we can see some of the ants of generation 25. Notice how two of the ants have found the “24” solution mentioned above, while some of them are still using the older “20”

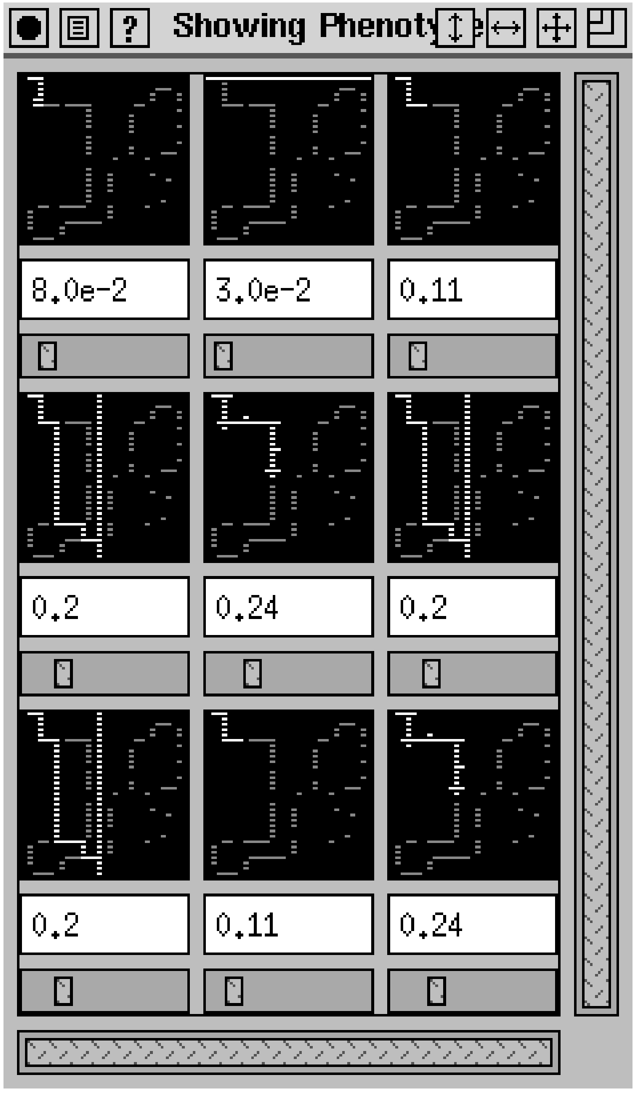

Figure 2: Evolved ants searching for food.

or “11” plans. There are also two very bad ants, where the good fitness of their parents has been lost due to mutation or unlucky crossover.

2 The Program

The program, “Gene”, is written to be easily extended with different data types representing problems you might want to solve with the genetic algorithm. “Gene” allows the user to choose:

- Crossover, mutation or both.
- Fitness provided by a predefined function or by the user. Experiments with neural nets learning the fitness concept have also been made, see section 4, “Neural networks”.
- Size of the population.
- How many generations to run in each step.
- If the genes should be displayed (using a predefined drawing function).

In every step, the maximum and average fitness values are displayed. You can also save and load the populations.

“Gene” is written in a lazy functional language called Haskell [1]. It uses a graphical user interface toolkit called Fudgets [5], and an extension of Haskell called PolyP [6], which enables you to write polytypic functions.

2.1 The user interface

The picture shows the main program while evolving the ants in the extended example above. The maximum and average fitness values are displayed at the top of the window. Notice that the program uses fitness values between zero and one, the values mentioned above are hence divided by 100 here.

The “Next” button will take us to the next generation, using crossover and/or mutation depending on what the user has chosen in the next column, “Mate”, “Mutant” or “Both”. The fitness values can be provided by the user using the scrollbars in the “Showing Phenotype” window, which you can see in the ant example, or by a function. Each individual’s fitness value is also shown in that window. The “Picture”, “Planning” and “Tree” buttons are used to choose between the different genes that can be evolved. The first two cases are described above, the “Tree” example uses a gene consisting of a list of 9 integer numbers to draw tree-like structures.

Figure 4 shows pictures evolved with the “Picture” example, using essentially the datatype `Expr` described above. The set of binary operations was extended with the “arctan” function, and unary operations “sinus” and “sqrt” were also used. These functions could of course have been evolved from the +,-,* and / operations, but more interesting pictures evolved faster with this extension.

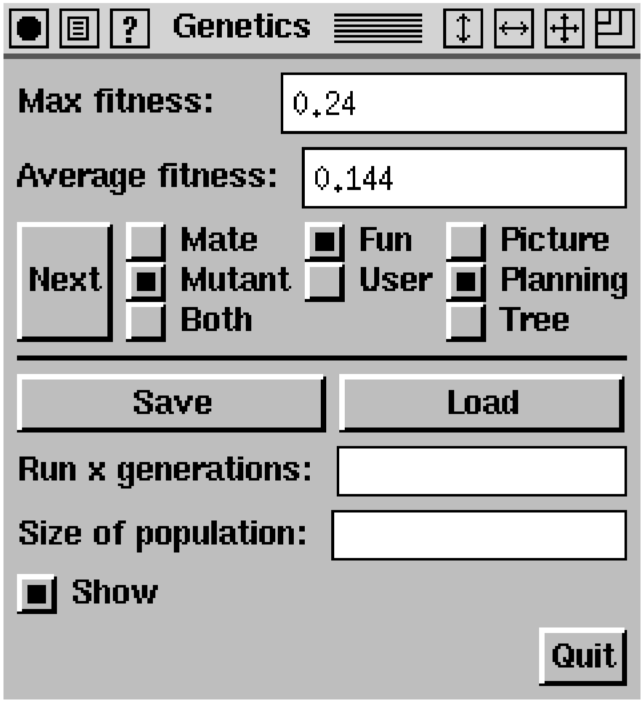

Figure 3: The graphical user interface.

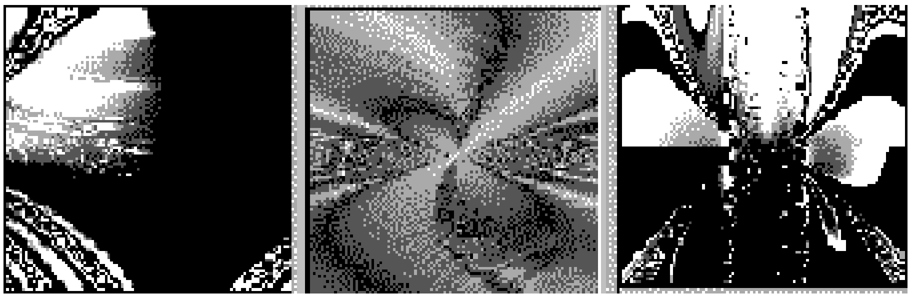

Figure 4: Evolved pictures.

3 Implementation

This section describes how various parts of the program were implemented. Problems and their solutions using the Haskell class system and polytypic programming will be covered.

3.1 A few words about Haskell

Haskell is a lazy functional language. The main building block is a function, which is defined by giving an appropriate expression.

```
square x = x*x
foo x y = if y == 0 then x else x / y
```

`Square` is a function taking a number and returning the square of that number. `Foo` is a protected divide function. The types of these functions are:

```
square :: Num a        => a -> a
foo    :: Fractional a => a -> a -> a
foo 1  :: Fractional a => a -> a
```

`Square` is defined on any type in the `Num` class, that is all numerical types such as `Int`, `Float` etc. It is a function taking one numerical value as its argument, and returning a value of the same type. `Foo` is defined on all types in the class `Fractional`, where the `/` function is defined. It takes two values as arguments and returns one value.

If we give `foo` one of its arguments, a new function will be returned taking only one argument. `Foo 1` is a function which returns the reciprocal of a number, `(foo 1) x = 1/x` if `x` is not zero. If a function is defined on many types, we use `a, b, c` etc to denote a type variable. The function length, giving the length of a list of values of any type, has the type `length :: [a] -> Int`, were `[a]` is a list of values of type `a`.

Values can be of many different types. There are predefined types, as `Int, Char, Bool` etc. Types can be tuples (e.g. `(a,b), (Int,Bool)`) or lists (e.g. `[a], [Char]`) of other types. There are function types as above (e.g. `Int -> Int`), and you can also define your own datatypes, as in the gene examples above.

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)

data Plan = If-food-ahead Plan Plan
          | Move
          | TurnLeft
          | TurnRight
          | Seq Plan Plan
```

A `Tree a` is either a Leaf with a value of type `a`, or `Branch t1 t2` were `t1` and `t2` are of type `Tree a`. To define functions on elements of these types, we use pattern matching.

```
tr = Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3) :: Tree Int

count :: Tree a -> Int
count t = case t of
            (Leaf x)       -> 1
            (Branch t1 t2) -> count t1 + count t2
```

`Count` returns the number of leafs in a tree. `Count tr` will evaluate to 3.

Observe that functions are values just as integers or trees. Notice also that since Haskell uses lazy evaluation, the values of the arguments to a function will not be computed until they are needed in the function. Consider the function `bar y x = x`, which takes two arguments and returns the second. `bar (1/0) 2` will be evaluated to 2, instead of giving a "divide by zero" error message as in a language with strict evaluation.

An important datatype is the list, defined as

```
data List a = Nil
            | Cons a (List a)
```

A list of the two numbers 1 and 2 is written `Cons 1 (Cons 2 Nil)`, or `[1,2]`. This shorter notation will be used in this text. The `Nil` case is written `[]`. There are several predefined functions operating on lists, such as

```
head :: List a -> a
tail :: List a -> List a
take :: Int -> List a -> List a
drop :: Int -> List a -> List a
map  :: (a -> b) -> List a -> List b
```

`Head` and `tail` give the first and all but the first elements of a list. `take n xs` returns the first n elements of xs, `drop n xs` all but the first n elements. The function `map` takes a function of type `a -> b` and a list of values of type `a` and applies the function to all the elements in the list.

For more information about Haskell, see [1] or [2].

3.2 The Gene class

As mentioned before, the program should work on several different datatypes. For every datatype used, there must be implementations of the corresponding fitness function, crossover function, drawing function etc. In the main program, we have to find out which type of genes that is currently in use and call the appropriate functions. A straight forward solution would be to define the general datatype Gene as

```
data Gene = Gene1 (List Float)
          | Gene2 (Expr)
          | Gene3 (Plan)
```

Expr and Plan refer to the examples above. Gene now describes an object that is either a list of floating point values, an expression or a plan. When the main program needs the fitness of an object, it has to test which kind of gene it is and then call the respective fitness function.

```
fitness g = case g of
              Gene1 l -> fitness_list l
              Gene2 e -> fitness_expr e
              Gene3 p -> fitness_plan p
```

The same construct has to be used when performing crossover, mutation etc.

But this is not a very nice solution. If we want to insert yet another type of genes or remove some of the old ones, the datatype Gene must be changed as well as all the "inner loops" of the main program. What we would like is to define fitness etc as overloaded functions, defined on several different data types. The program will then automatically find out what instance of the functions to call. Compare this with the member functions or methods of the objects in an object oriented language, where dynamic binding is used to call the appropriate method.

The Haskell class system provides a convenient way of doing this. A class GeneObj will be defined, containing declarations of all type specific functions. To use a datatype as a gene in the program, all you have to do is instantiating it as a GeneObj by defining the methods.

```
class GeneObj a where
  fitness   :: a -> Float
  crossover :: a -> a -> Float -> a
  mutant    :: a -> Float -> a
  draw      :: a -> Drawing
```

A type a is a member of the class GeneObj if the four functions above are defined. Fitness takes an object of type a and gives a floating point value. Crossover takes two objects of type a and a random number and returns a combination of the two objects. Mutant takes an object and a random number and returns a mutation of the object, and draw takes an object and returns a drawing.

```
instance GeneObj (Expr) where
  fitness g = ...
  crossover g1 g2 random = ...
  mutant g random = ...
  draw g = ...
```

If definitions of the methods are given, Expr becomes an instance of GeneObj. We can now apply for example the mutant function to any type which is a member of the class GeneObj, and the correct method will be used.

A population will be described as a list of genes. We want the population to have the same type for all different types of genes that can occur in it, so we define the population with an existentially qualified datatype as follows.

```
datatype Population = (GeneObj ?a) => Pop [?a]
```

If the type `?a` is a `GeneObj`, then a list of `?a` typed objects, `[?a]`, with the type constructor `Pop`, will have the type `Population` irrespective of the type `?a`. Observe that `?a` is not standard Haskell notation, but an extension in the HBC Haskell compiler, see [3]. See also Konstantin Läufers text on existential types [4].

```
e1 = BinOp '+' (Num 1) (Xcoord)  :: Expr
e2 = Ycoord                      :: Expr

p1 = If-food-ahead Move TurnLeft :: Plan
p2 = Seq TurnRight Move          :: Plan
```

`E1` and `e2` are of type `Expr, p1` and `p2` of type `Plan`.

```
pop1 = Pop [e1,e2] :: Population
pop2 = Pop [p1,p2] :: Population
```

`Pop1` and `pop2` are now of the same type, `Population`. The methods **`fitness`**, **`mutant`** etc can be used on all four objects. The main program will operate on populations, hence on the same type all the time, though the type of the genes in the populations may change during runtime.

Now, we can add or remove datatypes representing genes without having to change anything in the main program. But we still have to define all the methods for every datatype, even if the implementations might be essentially the same. This problem will be solved in the next section.

3.3 Polytypism

Consider the crossover function. In the picture example, a gene is an object of type `Expr`. To combine two genes, we want to replace a random sub-expression in the first gene with a random sub-expression from the second one. In the case of the food searching ant, the genes are of type `Plan`, and to perform crossover on two plans we want to replace a sub-plan with another sub-plan. It is obvious that these functions will be very similar, and we would like to generalise the concept of "replacing a sub-tree" so that we can write one single crossover function which operates on any datatype. This function will accept objects of any types as its arguments, and is hence called a *polytypic* function. (The term "sub-tree" will denote a sub-expression of any type. For example, the sub-tree of a list is itself a list.)

Note that the function "times", `(*) :: Num a => a -> a -> a`, also accepts arguments of many types. But this is a family of possibly unrelated functions, one for each instance in the `Num` class. The compilator uses type inference to determine which of these functions it should use. This is called a *polymorphic* or *overloaded* function. It is rather a set of functions using the same name than one function defined on many datatypes.

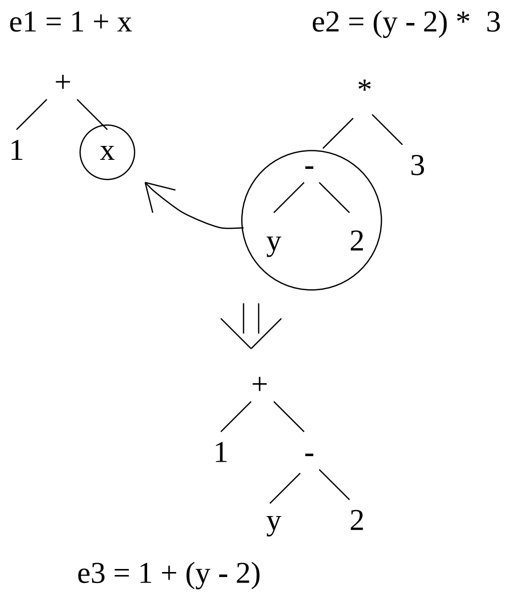

Figure 5: Genetic crossover.

The function `head :: [a] -> a` is also polymorphic, since the values in the list are never used in the definition of the function. It is not overloaded though, because there is only one definition of `head`.

A polytypic function is defined by induction on the structure of user-defined datatypes. There is only one definition, but instances for every datatype on which it is applied.

```
e1 = BinOp '+' (Num 1) Xcoord
 (e1 = 1 + x)
e2 = BinOp '*' (BinOp '-' Ycoord (Num 2)) (Num 3)
 (e2 = (y - 2) * 3)

p1 = If-food-ahead Move TurnLeft
p2 = Seq TurnRight Move
```

`Crossover e1 e2 rnd` and `crossover p1 p2 rnd` might evaluate to (depending on the random number)

```
e3 = BinOp '+' (Num 1) (BinOp '-' Ycoord (Num 2))
 (e3 = 1 + (y - 2))

p3 = If-food-ahead Move TurnRight
```

Where `e3` (`p3`) is `e1` (`p1`) with its right sub-tree replaced by `e2`'s (`p2`'s) left sub-tree. This is illustrated in figure 5.

We will use an extension of Haskell called PolyP to write these polytypic functions.

3.3.1 PolyP

PolyP is an extension of Haskell that allows us to define polytypic functions. A new kind of definition is introduced, the `polytypic` construct, which is used to define functions by induction over the structure of datatypes.

Recall that the datatype definitions can express constant types (e.g. `Int, Bool`), tuples, (e.g. `(Int,Bool)`, parameters as `a, b` etc, choices as in `T a = L a | B (T a) (T a)` and recursive types, as in `B (T a) (T a)`. The `polytypic` construct can handle all these cases.

We will now see descriptions of the polytypic functions which are used in the definitions of `mutant` and `crossover`. First of all, we need to convert a random floating point value between 0 and 1 to an integer in the range 0 to the size of a given object. This number will tell us which sub-tree to replace, or with which tree it should be replaced. To do this conversion, we need the polytypic `size` function, giving the number of type constructors in an object. The size of `e3 = BinOp '+' (Num 1) (BinOp '-' Ycoord (Num 2))` should be 5, since there are two `BinOp`, two `Num` and one `Ycoord` in `e3`.

```
size :: Regular d => d a -> Int
size = cata fsize

polytypic fsize :: f a Int -> Int
  = case f of
      Const t -> \x -> 1
      Empty   -> \x -> 1
      Par     -> \x -> 1
      Rec     -> \x -> x
      f + g   -> either fsize fsize
      f * g   -> \(x,y) -> (fsize x) + (fsize y)
```

(The class `Regular` consists of all datatypes such that they contain no function spaces, and the arguments on the left- and right hand side are the same.)

First, we define `size` as a *catamorphism*. A catamorphism is a function replacing all type constructors and parameters with functions and values. If we, in the datatype `Tree a`, replace `Branch` with the function `add` and `Leaf a` with 1, a tree as `t = Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)` will become `add (add 1 1) 1 = add 2 1 = 3`.

In the type declaration `size :: d a -> Int, d` is a type constructor as `Tree, Plan` or `List`. The type means that `size` takes a value of any regular datatype as its argument, and returns an integer.

The notation `\x -> e` denotes a function taking a value x and returning the value of the expression `e`. Hence `\x -> x` is the identity function, and `\x -> 1` a function returning 1 irrespective of its argument.

`Cata` takes as argument a function written with the `polytypic` construct, where a function is given for every possible way of constructing a datatype.

PolyP will use this defintion to generate appropriate functions for every datatype on which the `size` function will be applied.

```
data List a = Nil
            | Cons a (List a)

data Tree a = ...
```

`List a` has two cases, either it is an empty list `Nil` or a value of type `a` and another list. Hence, this type can be described as `Empty+(Par*Rec)`. Read + as "or", * as "and". A tree has the type `Par+(Rec*Rec)`. These types are the first argument to `fsize`. They are called *functors* of the datatypes. A value is converted to and from these abstract representations using the functions `inn :: f a (d a) -> d a` and `out :: d a -> f a (d a)`, where `f` (or `FunctorOf d`) is the functor of the datatype `d`. In the functions using the `polytypic` construct, the `f` in the type stands for `FunctorOf d`. The generated functions will essentially do the following:

```
size l = case l of
  Nil -> 1
  Cons a as -> 1 + (size as)

size t = case t of
  Leaf a -> 1
  Branch t1 t2 -> 1 + (size t1) + (size t2)
```

The type of `cata` is

```
cata :: Regular d => (FunctorOf d a b -> b) -> d a -> b
```

In the `Tree` case, the variable `d` will be `Tree`. The functor of `d`, will hence be `Par+(Rec*Rec)`. This functor takes two arguments, the type of the parameter and the type of the recursive parts. The first argument to `cata` is a function from this functor type to a type `b`, in the `size` example this function is `fsize :: FunctorOf d a Int -> Int`. The cases in `fsize` give the functions and values which `cata` uses to replace the constructors and parameters in an object.

A similar function is `para`, which expresses a *paramorphism*. It has the type

```
para :: Regular d => (d a -> FunctorOf d a b -> b) -> d a -> b
```

A paramorphism is similar to a catamorphism, but in the recursive step you can use not only the recursive result obtained from the children, but also the argument of the paramorphism. In the type of `para`, we can see that the first argument, a function as in `cata`, takes not only `f a b` as an argument but also an element of type `d a`.

3.3.2 The crossover and mutant functions

Now, we need a function that extracts the n:th sub-tree from a value. We define a function `scan`, which takes an element and returns a list of this element and all its sub-trees.

```
scan :: Regular d => d a -> [d a]
scan [1,2,3] = [[1,2,3],[2,3],[3],[]]

scan B (L 1) (L 2) = [(B (L 1) (L 2)),(L 1),(L 2)]

polytypic children :: f a [b] -> [b]
  = case f of
      Const t -> \x -> []
      Empty   -> \x -> []
      Par     -> \x -> []
      Rec     -> \x -> x
      f * g   -> \(x,y) -> children x ++ children y
      f + g   -> either children children

scan = para (\x -> \y -> x : children y)
```

We can now get any sub-tree by taking that element from this list.

The function `build` is the inverse of `scan`, it rebuilds the object from the scanned list. This function does not just take the first element of the list, instead it uses the type constructor, and eventually the parameter, from every item of the list.

```
 [[1,2,3],[2,3],[3],[]]
   -       -     -  --  => [1,2,3]

 [(B (L 1) (L 2)),(L 1),(L 2)]
   -               ---   ---   => B (L 1) (L 2)


build :: Regular d => [d a] -> d a
build xs = fst (build' xs)

build' = \(x:xs) ->
  let a = case xs of
            [] -> fbuild [] (out x)
            xs -> let b = build xs
                  in fbuild (fst b : (snd b)) (out x)
  in (inn (fst a),snd a)

inL x = (Left (fst x),snd x)
inR x = (Right (fst x),snd x)

polytypic fbuild :: [b] -> f a b -> (f a b,[b])
  = \bs -> case f of
      Const t -> \x -> (x,bs)
      Empty -> \x -> ((),bs)
      Par   -> \x -> (x,bs)
```

```
Rec   -> \x -> (head bs,tail bs)
f * g -> \(x,y) -> let f1 = fbuild bs x
                       f2 = fbuild (snd f1) y
                   in ((fst f1,fst f2),snd f2)
f + g -> \x -> case x of
                 Left x  -> inL (fbuild bs x)
                 Right x -> inR (fbuild bs x)
```

If we replace some of the items in the list before building it, the new object will have some of its parts replaced.

```
build ((take 2 (scan (B L1 L2))) ++ (scan (B L3 L4))) =
build ([B L1 L2, L1] ++ [B L3 L4, L3, L4]) =
B L1 (B L3 L4)
```

where Lx is short for (L x).

Now we can define the function `replace`, taking two random numbers and two objects and returning one object.

```
replace :: Regular d => Float -> Float -> d a -> d a -> d a
replace r s x y = let r' = truncate(r*size x)
                      s' = truncate(s*size y)
  in build (take r' (scan x) ++
            scan (head (drop s' (scan y))) ++
            drop (r'+1) (scan x))

e1 = BinOp '+' (Num 1) Xcoord
e2 = BinOp '*' (BinOp '-' Ycoord (Num 2)) (Num 3)
replace 0.4 0.1 e1 e2 =
  BinOp '+' (BinOp '-' Ycoord (Num 2)) Xcoord
```

`Replace` can now be used as crossover function, and also as a mutant function if the fourth argument is a random value of type `d a` and the second argument is zero.

3.3.3 Problems with PolyP

As mentioned before, for every polytypic function you write PolyP has to generate a specific function for every type on which you want to apply the function. To do this, PolyP must have access to all your code to see where and on what types of values you use your functions. This may cause trouble, since the code can't be divided into different modules and since the current version of PolyP can only handle a subset of Haskell.

A bigger problem is that we used the existentially qualified type `Population :: (GeneObj ?a) => Pop [?a]` to express that the individuals in a population can be of any type in the class `GeneObj`. When PolyP sees that `size` or `replace` are applied to objects in the list of a population, there is no way of determining the type of those objects. This type is hidden by the `Pop` constructor. Hence, we can not use our polytypic functions on the populations.

A solution to these problems is to use the polytypic functions only inside the definitions of the methods of the class `GeneObj`. There, the type of the objects is known. To avoid giving our whole program to PolyP, we can create a "fake" main program which just calls the functions with arguments of appropriate types. In this example, we want to use the previously defined types `Expr` and `Plan`.

```
e :: Expr
p :: Plan

main = ((size e,size p),(replace 0 0 e e,replace 0 0 p p))
```

PolyP now generates the functions `size_Expr, size_Plan, replace_Expr` and `replace_Plan`. Unfortunately, the polytypism is now gone since these functions are not at all polytypic. The abstraction PolyP was supposed to provide is lost, since we instead of having one single function for use on any datatype ended up with one function for each datatype.

(In future versions of PolyP, it will probably be possible to give a list of the datatypes you want to use for every polytypic function, instead of giving a main function.)

```
mutant' g rnd mkFun repFun = let g' = mkFun rnd
  in repFun rnd 0 g g'

instance GeneObj (Expr) where
  fitness = ...
  crossover g1 g2 rnd = replace_Expr rnd rnd g1 g2
  mutant g rnd = mutant' g rnd mkExpr replace_Expr
  draw g1 = ...

instance GeneObj (Plan) where
  fitness = ...
  crossover p1 p2 rnd = replace_Plan rnd rnd p1 p2
  mutant p rnd = mutant' p rnd mkPlan replace_Plan
  draw p1 = ...
```

(The size functions should also be provided as arguments to replace.)

Now we have restored this lost abstraction, and created the new overloaded functions `crossover` and `mutant`. These functions can be applied to any object in the `GeneObj` class, and hence on the elements in an object of type `Population`.

To insert a new type of genes in the program, all that must be done now is defining an abstract datatype and giving appropriate fitness and drawing functions. If there is no natural fitness value for a gene, you can let the user provide these values using the graphical user interface. The drawing function can create a drawing using the genes as in the picture or ant examples above, or just print the gene.

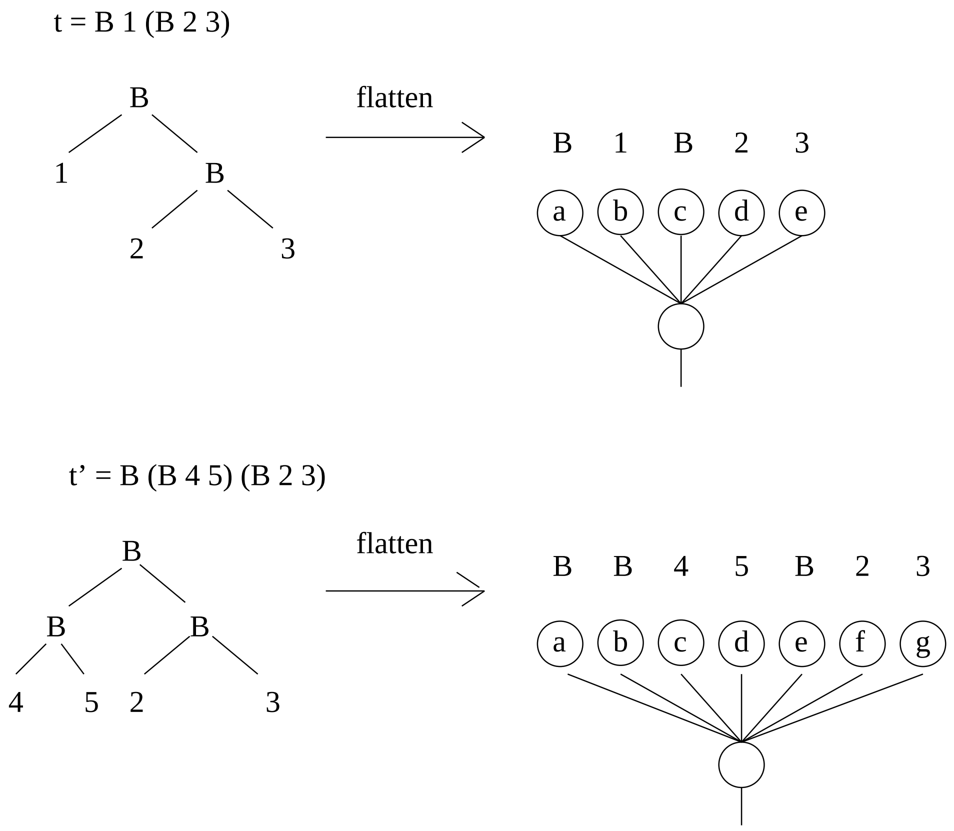

Figure 6: A neural network.

4 Neural networks

If there is no natural fitness function for a problem, as in the picture example above, the user has to provide the fitness of each gene using scrollbars. This might be very time consuming, and we would like the program to learn the concept of fitness by looking at the choices the user makes.

A two layer feed forward neural network was tried as a solution of this problem. A network consists of a number of neurons with internal weights. Each neuron gets a number of input values and outputs a single value, which is a weighted sum of the input. By giving the network the correct or expected answer of a given input vector, the weights can be adjusted using an error back propagation method. When the weight values have converged, the network will hopefully give correct output values.

Unfortunately, this method is appropriate only if the input is of constant size, as in the insect drawing example described above. If the size changes during runtime, as in the other two examples, we could of course extend the network with new neurons, but these will not have the correct weights.

For example, consider the case when the genes have a tree structure as in the picture and planning examples above. To input a gene in the network, we first have to flatten it in some way, generate a list of values representing the tree. Assume now that the neurons have adapted correct weights for evaluating a good right sub-tree. But when the left sub-tree changes due to mutation or crossover, the right tree will appear further to the right in the list, and hence be input to a new set of neurons. The learned information will be lost.

Observe that in figure 6, all values in the top rows are input to all neurons in the first layer. The object `t` has a very good right sub-tree, `B 2 3`. This will be input number 3, 4 and 5 to the first layer. The neurons a through e have learned to evaluate this structure as a good one, they have weights number 3, 4 and 5 well adjusted. But when the left sub-tree `1` is replaced with `B 4 5`, the right tree will be evaluated with weights number 5, 6 and 7. This network will not be able to tell us that `B 2 3` is good.

There are probably solutions to this problem, but since the learning time and convergence behaviour was not very good even for genes of constant size, no further attempts were made.

4.1 Implementation of networks

To implement a neural network, you have to represent the weights of each neuron as an internal state in the program. In a functional language, you can do this by describing the network as a state transforming function. It takes a vector of input values and a state, and returns the output of the network paired with a new state.

The type of this function is

```
net :: NetInput -> S NetOutput

type S a = State -> (a,State)

data NetInput = NInput [Float]
              | NLearn [Float] Float

type NetOutput = Float

data State = [[Float]]
```

The function `net` takes either a list of input values or a list of values together with the correct output value. The latter case is used in the learning phase. The result is of type `S NetOutput`, which is a function taking a state and returning a state paired with the output. The state is a list of lists of values, one list of weights for each neuron in the network.

When `NInput xs` is input to the net, the corresponding output will be calculated using the weights in the state, and the new state will be equal to the previous. In the `NLearn xs c` case, the same output will be computed and then compared to the expected output `c`. The error is used to compute the adjusted weights of the new state.

```
net (NInput xs)   = \s -> (output xs s,s)
net (NLearn xs c) = \s -> let o = output xs s
```

```
                          in (o,update s (o-c))

output :: [Float] -> State -> Float
update :: State -> Float -> State

learn 0 s = fst (net (NInput [1,2,3]) s)
learn x s = learn (x-1) (snd (net (NLearn [1,2,3] 4) s))
```

To learn that the output of `[1,2,3]` should be four, we can use the function `learn :: Int -> State -> NetOutput`, which takes the number of training steps and an initial state as arguments.

For further details on neural nets, see [7].

5 Conclusions

As soon as you start experimenting with genetic algorithms, you realize that it is a powerful tool for finding solutions to very different problems. This program may not be a very good implementation of genetic algorithms, for example you should be able to adjust the mutation rate. But if you just want to try the method on a problem, it is very easy to do. Further investigations in the area of learning fitness functions would be interesting.

The use of polytypism makes the program easier to change, and increases the possibility of reusing your code. Every time you feel that you are writing esentially the same code over and over again for use on different datatypes, you should consider using a language with polytypism, as Haskell with PolyP.

6 Appendix A, time disposal

- Research on genetic algorithms, approx. 2 weeks.
- Implementing of main program, approx. 3 weeks.
- Implementing the examples, approx. 4 weeks.
- Research on neural networks, approx. 1 week.
- Implementing network, approx. 1 week.
- Research on polytypic programming, approx. 2 weeks.
- Implementing polytypic functions, approx. 2 weeks.
- Writing the report, approx. 5 weeks.

7 Appendix B, Program code

Please note that the example code in the previous chapters sometimes have been slightly simplified. This is the code which is actually used in the program.

7.1 The Main program

```
import AllFudgets
import Class
import Misc
import Array
import Net1
import Time
import Expr

type Grey = Array Int Pixel

type Fits = Array Int Float

data State = State [Int] Fits NState Population Mating Choose

data GOutput = GoShow Int Population
             | GoFit Int Float
             | GoDisp Bool
             | GoSize Int
             | GoMax (Float,Float)
             | GoPoint Point
             | GoPrint String

data GFitn = GFitn Int Float

data Mating = Mate
            | Mutant
            | Both
     deriving Eq

data Choose = Function
            | User
            | Net
            | Learn
     deriving Eq

data Method = Method Mating Choose

data GInput = GiNext Int
            | GiMeth Mating
```

```
           | GiChos Choose
           | GiNew  Population
           | GiDisp Bool
           | GiSize Int
           | GiPrint
           | GiLoad Population

main = do
        t <- getClockTime
        fudlogue (shellF "Genetics"
                   (mainF' (getSeed t) >*<
                    (quitButtonF >==< nullF)))
  where getSeed = fromInteger . ctPicosec . toUTCTime

mainF' s = loopLeftF
           (dispPopupF 9 >==<
            mapstateF eval (startstate 9) >==<
            idF >+< buttonsF
           )
  where
  startstate x =
    State (randomInts s s) (listArray (1,x) [0,0..])
           nstate (mkGenes g0 x s) Mutant User
  nstate = NState (replicate 3 0) --(length (flat g0)+1)

parse1 :: String -> [String]
parse1 s = let (ns,os) = head (reads s) in
           if os=="" then ns else parse1 os

readGenesF :: F (Either Click GInput) GInput
readGenesF = mapstateF readG (head genes) >==<
             ((filterRightSP >^^=<
               (nullF >+< mapF parse1) >==<
               snd >^=< readFileF >==<
               snd >^=< stringPopupF "" >=^<
               (\Click -> (Just "Filename:",Nothing))
              ) >+< idF
             )
  where
  readG :: Population -> (Either [String] GInput) ->
           (Population,[GInput])
  readG g@(Pop (g':_)) (Left es) =
    (g,[GiLoad (Pop (map (setval g') es))])
  readG _ (Right (GiNew g)) = (g,[])
  readG g _ = (g,[])

printF :: F String a
```

```
printF =
  dynF nullF >==<
  mapstateF writeG ("","") >==<
  (snd >^=< stringPopupF "") >+< idF >==<
  putlistF (\s -> [Left (Just "Filename:",Nothing),Right s])
  where
  writeG (n,_) (Right s) =
    if n /= ""
    then (("",""),[Right s])
    else ((n,s),[])
  writeG (_,s) (Left n)  =
    if s /= ""
    then ((n,s),[Left (outputF n),Right s])
    else ((n,s),[Left (outputF n)])

buttonsF :: F GInput GInput
buttonsF = loopF $ placerF (revP verticalP) $
  (GiDisp >^=< startupF [False]
    (spacerF leftS (toggleButtonF "Show")) >==< nullF) >*<
  (GiSize >^=< inputDoneSP >^^=<
    ("Size of population: " `labLeftOfF` intF) >==< nullF) >*<
  (GiNext >^=< inputDoneSP >^^=<
    ("Run x generations: " `labLeftOfF` intF) >==< nullF) >*<
  placerF horizontalP
    ((const GiPrint >^=< buttonF "Save" >==< nullF) >*<
     (readGenesF >==< ((buttonF "Load" >==< nullF) >+< idF) >=^<
      Right)) >*<
  graphicsLabelF (hFiller 2) >*<
  (untaggedListLF horizontalP
  [(\_ -> GiNext 1) >^=< buttonF "Next" >==< nullF,
   GiMeth >^=< radioGroupF
     [(Mate,"Mate"),(Mutant,"Mutant"),(Both,"Both")]
       Mutant >==< nullF,
   GiChos >^=< radioGroupF
     [(Function,"Fun"),(User,"User")] User >==< nullF,
   GiNew . head . flip drop genes >^=< radioGroupF
     [(1,"Picture"),(2,"Planning"),(3,"Tree")] 1 >==< nullF
  ] >==< nullF)
    >=^^< filterSP (\x -> case x of
                          GiNew _ -> True
                          _ -> False})

displayGeneF :: Grey -> Point ->
  F (Either Float Population) Float
displayGeneF grey s =
  filterLeftSP >^^=< placerF (revP verticalP)
    ((scF >==< dF) >+< gF)
```

```
  where
  scF = (\(x,_,_) -> (fromInt x)/100.0) >^=<
     startupF [ResizePot 10 110] (hPotF Nothing) >=^<
     (MovePot . fromInt . round . ((*) 100.0))
  gF = nullF >==< showF grey s
  dF = (displayF >=^< show) >*< idF

showF :: Grey -> Point -> F Population GfxOutput
showF grey s =
  spacerF (sizeS s) (graphicsDispF >=^<
          (\(Pop (g:_)) -> replaceAllGfx
            (atomicD (FD (picsize g) (toGfx g grey))))

displayerF :: Int -> Point ->
  F (Int,Either Float Population) (Either GFitn b)
displayerF x s =
  allocGreyScale $ \grey ->
  shellF "Showing Phenotypes" $
  scrollF $
  (\(x,y) -> Left (GFitn x y)) >^=<
  placerF ((matrixP . round . sqrt . fromInt) x)
  (listF $ zip [1..x] (replicate x (displayGeneF grey s)))

dispPopupF :: Int -> F GOutput (Either GFitn a)
dispPopupF x = dynF nullF >==<
               filterRightSP >^^=<
               ((maxavF >+< printF) >+<
                mapstateF dyneval (False,x,Point 0 0)) >=^<
               (\x -> case x of
                 GoMax (m,a) -> Left (Left (m,a))
                 GoPrint s -> Left (Right s)
                 y         -> Right y
               )
  where
  dyneval (_,i,p) (GoDisp b)  =
    ((b,i,p),[Left (if b then displayerF i p else nullF)])
  dyneval (b,_,p) (GoSize i)  =
    ((b,i,p),if b then [Left (displayerF i p)] else [])
  dyneval (b,i,_) (GoPoint p) =
    ((b,i,p),if b then [Left (displayerF i p)] else [])
  dyneval (False,i,p) _       = ((False,i,p),[])
  dyneval b (GoShow i g)      = (b,[Right (i,Right g)])
  dyneval b (GoFit i f)       = (b,[Right (i,Left f)])
  maxavF = placerF verticalP
    (labLeftOfF "Max fitness:" (displayF >=^< (show . fst)) >*<
     labLeftOfF "Average fitness: " (displayF >=^< (show . snd)))
```

```
sr1 (State _ fa ns gs m c) ss = State ss fa ns gs m c
sr2 (State ss _ ns gs m c) fa = State ss fa ns gs m c
sr3 (State ss fa _ gs m c) ns = State ss fa ns gs m c
sr4 (State ss fa ns _ m c) gs = State ss fa ns gs m c
sr5 (State ss fa ns gs _ c) m = State ss fa ns gs m c
sr6 (State ss fa ns gs m _) c = State ss fa ns gs m c

gosize :: Population -> Point
gosize (Pop ps) = picsize (head ps)

eval :: State -> Either GFitn GInput -> (State,[GOutput])
eval s@(State ss fa ns gs@(Pop gs') m c) x =
  (case x of
    Left (GFitn i f) -> let nfa = fa//[(i,f)]
      in (sr2 s nfa,[GoFit i f])
    Right GiPrint      -> (s,[GoPrint (show (map value gs'))])
    Right (GiLoad ngs@(Pop ngs'')) ->
      let nfa = newfit ngs c
          nns = NState
            (replicate (length (flat (head ngs'')) + 1) 0)}
      in (sr2 (sr3 (sr4 s ngs) nns) nfa,
        GoSize (length ngs'') :
          GoPoint (gosize ngs) : outgen ngs nfa)
    Right (GiDisp b)   ->
      (s,GoPoint (gosize gs) : GoDisp b : outgen gs fa)
    Right (GiMeth m')  -> (sr5 s m',[])
    Right (GiChos c')  -> let nfa = newfit gs c'
      in (sr2 (sr6 s c') nfa,outfit nfa)
    Right (GiNew g)    ->
      let x = snd (bounds fa)
          ngs = mkGenes g x (head ss)
          nfa = newfit ngs c
          nns = NState (replicate 3 0)} --(length (flat g1) + 1)
      in (sr1 (sr2 (sr3 (sr4 s ngs) nns) nfa) (tail ss),
          GoPoint (gosize ngs) : outgen ngs nfa)
    Right (GiNext x)   ->
      let (nss,ngs,nfa,nns) = next x (ss,gs,fa,ns)
      in (sr1 (sr2 (sr3 (sr4 s ngs) nns) nfa) nss,
          outgen ngs nfa)
      where
      next 0 sgfn = sgfn
      next y (s,g@(Pop g'),f,n) =
        next (y-1) (tail s,ng,newfit ng c,nn)
        where
        ng = mkGeneration (head s) m c f nn g (snd $ bounds f)
        nn =
          (case c of
```

```
          Learn ->
            snd (neteval (NLearns (map flat g') (elems f)) n)
          _     -> n)
     Right (GiSize x)   ->
       let ngs = mkGeneration (head ss) m c fa nns gs x
           nfa = newfit ngs c
           nns =
             (case c of
              Learn ->
                snd (neteval (NLearns (map flat gs')
                     (elems fa)) ns)
                _     -> ns)
       in (sr1 (sr2 (sr3 (sr4 s ngs) nns) nfa) (tail ss),
           GoSize x : outgen ngs nfa)
   )
   where
   outfit :: Fits -> [GOutput]
   outfit a =
    GoMax (maximum (elems a), sum (elems a) /
           fromInt (snd (bounds a))) :
           map (\(x,y) -> GoFit x y) (assocs a)
   newfit :: Population -> Choose -> Fits
   newfit (Pop gs') c' = let l = length gs' in
     case c' of
       Function -> listArray (1,l) (map fitness gs')
       User     -> listArray (1,l) (replicate l 0)
       Net      ->
         listArray (1,l) (fst (neteval (NInput (map flat gs')) ns))
       Learn    ->
         listArray (1,l) (replicate l 0) --(map fitness gs')
   outgen :: Population -> Fits -> [GOutput]
   outgen (Pop gs') a = outfit a ++
     zipWith (\x y -> GoShow x (Pop [y])) [1..] gs'

mkGeneration :: Int -> Mating -> Choose -> Fits -> NState ->
   Population -> Int -> Population
mkGeneration seed m c fa ns (Pop gs) nr =
   mkGen nr (randomFloats seed (seed+1)) []
   where
   fitness' x = let y = fitness x
     in y*y
   fits = map (\x -> x*x) (elems fa)
   gsum = case c of
            Function -> foldl (\x g -> x + fitness' g) 0 gs
            _        -> sum fits
   mkGen 0 _ ms = Pop ms
   mkGen x (r1:r2:r3:r4:rs) ms = mkGen (x-1) rs (mate:ms)
```

```
    where
    mate =
      case m of
      Mate   ->
        love (getG (r1*gsum) gs fits) (getG (r2*gsum) gs fits) r3
      Mutant -> mutant (getG (r1*gsum) gs fits) r2
      Both   ->
        mutant (love (getG (r1*gsum) gs fits) (getG (r2*gsum)
                gs fits) r3) r4
    getG y gs' fs = case c of
      Function -> getGf y gs'
      _        -> getG_ y gs' fs
    getGf _ (g':[]) = g'
    getGf y (g':gs') = if y < fitness' g'
                       then g'
                       else getGf (y - fitness' g') gs'
    getG_ _ (g':[]) _ = g'
    getG_ y (g':gs') (f:fs) = if y < f
                              then g'
                              else getG_ (y - f) gs' fs
```

7.2 The Gene class

```
module Class where

import Misc
import Expr
import Plan
import Polyps
import Array
import AllFudgets

class GeneObj a where
  value   :: a -> String
  setval  :: a -> String -> a
  fitness :: a -> Float
  love    :: a -> a -> Float -> a
  mutant  :: a -> Float -> a
  mkGene  :: a -> [Int] -> (a,[Int])
  picsize :: a -> Point
  toGfx   :: a -> (Array Int Pixel) -> [DrawCommand]
  flat    :: a -> [Float]

data Gene1 = Gene1 Int Int
  deriving (Show,Read)
```

```
instance GeneObj (Gene1) where
  value g = show g
  setval g s = read s
  fitness (Gene1 x _) = fromInt x / 100
  love (Gene1 x x') (Gene1 y y') rnd =
    if rnd < 0.5 then (Gene1 x y') else (Gene1 x' y)
  mutant (Gene1 x y) rnd = Gene1 y x
  mkGene _ (x:y:xs) = (Gene1 (mod x 100) (mod y 100),xs)
  picsize _ = Point 100 20
  toGfx (Gene1 x y) _ = [DrawString (Point 10 10) (show (x,y))]
  flat (Gene1 x y) = [fromInt x / 100,fromInt y / 100]

data GPic = GPic (Expr Float)
  deriving (Show,Read)

newrnd rnd = (sin (rnd*100)) / 2 + 0.5

instance GeneObj (GPic) where
  value g = show g
  setval _ s = read s
  fitness (GPic e) = fromInt (size_Expr e) / 1000
  love (GPic e) (GPic e') rnd =
    GPic (rep repl_Expr size_Expr rnd (newrnd rnd) e e')
  mutant (GPic e) rnd =
    let s = round (rnd*999999)
        (e',_) = mkExpr (randomInts s (s*2))
    in GPic (rep repl_Expr size_Expr rnd 0 e e')
  mkGene _ xs = let (e,ss) = mkExpr xs
    in (GPic e,ss)
  toGfx (GPic e) grey =
    [CreatePutImage (Rect origin (Point 100 100)) zPixmap pix]
    where
    cut x = if x < 0 then 0 else if x > 0.96 then 0.96 else x
    pix = [grey!pixels x y | x <- [0 .. 99], y <- [0 .. 99]]
    pixels x y =
      let p = xyeval e ((fromInt x)/50-1) ((fromInt y)/50-1)
      in truncate
         ((if p<0 then 0 else if p>0.95 then 0.95 else p)*
           fromInt greyLevels)
  flat _ = []
  picsize _ = Point 100 100

data GPlan = GPlan (Plan Int) (Int,([(Int,Int)],[(Int,Int)]))
  deriving (Read,Show)

instance GeneObj (GPlan) where
  value (GPlan p _) = show p
```

```
  setval _ s = let np = read s
    in GPlan np (pGfx np)
  fitness (GPlan p (sc,_)) = fromInt sc / 100
  love (GPlan p _) (GPlan p' _) rnd =
    let np = rep repl_Plan size_Plan rnd (newrnd rnd) p p'
    in GPlan np (pGfx np)
  mutant (GPlan p _) rnd =
    let s = round (rnd*999999)
        (p',_) = mkPlan (randomInts s (s*2))
        np = rep repl_Plan size_Plan rnd 0 p p'
    in GPlan np (pGfx np)
  mkGene _ xs = let (p,ss) = mkPlan xs
    in (GPlan p (pGfx p),ss)
  toGfx (GPlan _ (_,(es,fs))) grey =
    [CreatePutImage (Rect origin (Point 64 64)) zPixmap pix]
    where
    gl = div greyLevels 2
    pix0 = (grey!0)
    pix1 = (grey!gl)
    pix2 = (grey!(gl+gl-1))
    pix = pixels [(x,div y 2) |
                  y <- [0 .. 63], x <- [0 .. 31]] es fs
    pixels [] _ _ = []
    pixels (x:xs) es fs =
      if es/=[] && x==head es
      then pix2:pix2: pixels xs (tail es)
           (if fs/=[] && x==head fs then tail fs else fs)
      else if fs/=[] && x==head fs
           then pix1:pix1: pixels xs es (tail fs)
           else pix0:pix0: pixels xs es fs
  flat _ = []
  picsize _ = Point 64 64

data Insect = Insect [Int]
  deriving (Read,Show)

instance GeneObj (Insect) where
  value g = show g
  setval _ s = read s
  fitness _ = 0
  love (Insect i) (Insect i') rnd =
    Insect (rep repl_Ins size_Ins rnd rnd i i')
  mutant gi@(Insect i) rnd =
    let s = round (rnd*999999)
        (Insect i') = fst (mkGene gi (randomInts s (s*2)))
    in Insect (if rnd < 0.5
               then rep repl_Ins size_Ins rnd rnd i i'
```

```
                else rep repl_Ins size_Ins rnd rnd i' i)
  mkGene _ xs =
    (Insect (map (\x -> (mod x 9) - 4) (take 9 xs)),drop 9 xs)
  toGfx (Insect gs) _ =
    lcenter (map DrawLine (mktree 50 50 2 ((gs!!8) + 5)))
    where
    size = 3
    dx = listArray (0,7)
      [-gs!!1,-gs!!0,0,gs!!0,gs!!1,gs!!2,0,-gs!!2]
    dy = listArray (0,7)
      [gs!!5,gs!!4,gs!!3,gs!!4,gs!!5,gs!!6,gs!!7,gs!!6]
    mktree x y dir len =
      let ndir = mod dir 8
          xnew = x + div (len * (dx!ndir) * size + 2) 4
          ynew = y + div (len * (dy!ndir) * size + 2) 4
      in if len == 0 then [] else lL x y xnew ynew :
            mktree xnew ynew (ndir-1) (len-1) ++
            mktree xnew ynew (ndir+1) (len-1)
  flat (Insect i) = map (((*) 0.1) . fromInt) i
  picsize _ = Point 100 100

lcenter :: [DrawCommand] -> [DrawCommand]
lcenter ds =
  map (flip moveDrawCommand
       (Point 0 (getcenter (getedges ds (50,50))))) ds
  where
  getcenter (y1,y2) = (div (y1+100-y2) 2) - y1
  getedges :: [DrawCommand] -> (Int,Int) -> (Int,Int)
  getedges [] (y1,y2) = (y1,y2)
  getedges (DrawLine (Line (Point _ y1) (Point _ y2)):ds)
    (y1',y2') = getedges ds (min y1 y1',max y2 y2')
  getedges (_:ds) yp = getedges ds yp

data Population = (GeneObj ?a) => Pop [?a]

mkGenes :: Population -> Int -> Int -> Population
mkGenes (Pop (g1:_)) nr rnd =
  mkGenes' nr (randomInts rnd (rnd*2)) []
  where
  mkGenes' 0 _ gs = Pop gs
  mkGenes' n rs gs = let (g2,rs') = mkGene g1 rs
    in mkGenes' (n-1) rs' (g2:gs)

g0 = Pop [Gene1 0 0]
g1 = Pop [GPic (Var 'X')]
g2 = Pop [GPlan Move (0,([],[]))]
g3 = Pop [Insect [0 .. 8]]
```

```
genes :: [Population]
genes = [g0,g1,g2,g3]
```

7.3 The Picture example

```
module Expr where

import Polyps

mkVar x = if odd x then 'X' else 'Y'

mkbOp :: Int -> Char
mkbOp x = case mod x 5 of
  0 -> '*'
  1 -> '/'
  2 -> '+'
  3 -> '-'
  4 -> 'a'

mkuOp x = case mod x 2 of
  0 -> 's'
  1 -> 'r'

getbOp :: Char -> (Float -> Float -> Float)
getbOp o = case o of
  '*' -> (*)
  '/' -> (\ x y -> if y==0 then x/0.001 else x/y)
  '+' -> (+)
  '-' -> (-)
  'a' -> (\ x y -> if x==0 && y==0 then 0 else atan2 y x)

getuOp o = case o of
  's' -> sin
  'r' -> sqrt . abs

mkExpr :: [Int] -> (Expr Float,[Int])
mkExpr (s':s:ss) = case mod s' 5 of
  0 -> let r = fromInt s * 4.6566130638969828e-10
       in (Num (r*2-1),ss)
  1 -> (Var 'Y',ss)
  2 -> (Var 'X',ss)
  3 -> let (e1,ss1) = mkExpr ss
       in (Uop (mkuOp s) e1,ss1)
  4 -> let (e1,ss1) = mkExpr ss
           (e2,ss2) = mkExpr ss1
```

```
      in (Bop (mkbOp s) e1 e2,ss2)
  5 -> let (e1,ss1) = mkExpr ss
           (e2,ss2) = mkExpr ss1
           (e3,ss3) = mkExpr ss2
      in (Iflte e1 e2 e3,ss3)

xyeval :: Expr Float -> Float -> Float -> Float
xyeval (Num f) _ _          = f
xyeval (Var 'X') x _        = x
xyeval (Var 'Y') _ y        = y
xyeval (Bop o e1 e2) x y    =
  (getbOp o) (xyeval e1 x y) (xyeval e2 x y)
xyeval (Uop o e1) x y       = (getuOp o) (xyeval e1 x y)
xyeval (Iflte e1 e2 e3) x y = if (xyeval e1 x y) <= 0
                              then (xyeval e2 x y)
                              else (xyeval e3 x y)
```

7.4 The Planning example

```
module Plan where

import Array
import Polyps

data Pos = Pos Int Int
  deriving (Eq)

type Dir = Int

data FState = FState Int Pos Dir [(Int,Int)] Int

data PState = PState Int Pos Dir [(Int,Int)] [(Int,Int)] Int

mkPlan :: [Int] -> (Plan a,[Int])
mkPlan ss = fst (mkPlan' ss 0)
  where
  mkPlan' (s:ss) x = case (if x>5 then mod s 3 else mod s 5) of
    0 -> ((Move,ss),x+1)
    1 -> ((TurnLeft,ss),x+1)
    2 -> ((TurnRight,ss),x+1)
    3 -> let ((p1,ss1),x1) = mkPlan' ss (x+1)
             ((p2,ss2),x2) = mkPlan' ss1 x1
         in ((Progn2 p1 p2,ss2),x2)
    4 -> let ((p1,ss1),x1) = mkPlan' ss (x+1)
             ((p2,ss2),x2) = mkPlan' ss1 x1
         in ((Iffood p1 p2,ss2),x2)
```

```
move (Pos x y) d = case d of
  0 -> Pos x (mod (y-1) 32)
  1 -> Pos (mod (x+1) 32) y
  2 -> Pos x (mod (y+1) 32)
  3 -> Pos (mod (x-1) 32) y

foodList =
  [(1,0), (2,0), (3,0), (3,1), (3,2), (25,2), (26,2), (27,2),
   (3,3), (24,3), (29,3), (3,4), (24,4), (29,4), (3,5), (4,5),
   (5,5), (6,5), (8,5), (9,5), (10,5), (11,5), (12,5), (21,5),
   (22,5), (12,6), (29,6), (12,7), (20,7), (12,8), (20,8), (12,9),
   (20,9), (29,9), (20,10), (12,11), (12,12), (29,12), (12,13),
   (20,13), (12,14), (20,14), (26,14), (27,14), (28,14), (17,15),
   (23,15), (12,17), (12,18), (16,18), (24,18), (12,19), (16,19),
   (27,19), (12,20), (16,20), (12,21), (16,21), (12,22), (12,23),
   (26,23), (3,24), (4,24), (7,24), (8,24), (9,24), (10,24),
   (11,24), (16,24), (23,24), (1,25), (16,25), (1,26), (16,26),
   (1,27), (8,27), (9,27), (10,27), (11,27), (12,27), (13,27),
   (14,27), (1,28), (7,28), (7,29), (2,30), (3,30), (4,30), (5,30)]

foodArray =
  listArray (0,1023) [if elem (x,y) foodList
                      then 1
                      else 0 | y <- [0..31], x <- [0..31]]

food' :: Pos -> [(Int,Int)] -> Bool
food' (Pos x y) fa = elem (x,y) fa

eat :: Pos -> [(Int,Int)] -> (Bool,[(Int,Int)])
eat _ [] = (False,[])
eat p@(Pos x y) ((f@(fx,fy)):fs) = if x==fx && y==fy
                                   then (True,fs)
                                   else let (b,fs') = eat p fs
                                        in (b,f:fs')

pGfx :: Plan a -> (Int,([(Int,Int)],[(Int,Int)]))
pGfx plan = (sc,(vs,foodList))
  where
  (PState sc _ _ _ vs _) =
    pFit' (PState 0 (Pos 0 0) 1 foodList [] 200)
  pFit' s@(PState _ _ _ _ _ t) = if t <= 0
                                 then s
                                 else pFit' (pFit'' s plan)
  pFit'' s@(PState sc p d ps vs t) pl = if t <= 0 then s else
    case pl of
      Iffood p1 p2 -> pFit'' s (if food' (move p d) ps
```

```
                                then p1
                                else p2)
      Progn2 p1 p2 -> let s' = pFit'' s p1
                      in pFit'' s' p2
      Move         -> let p'@(Pos x y) = move p d
                          (b,ps') = eat p' ps
                      in (PState (if b
                                  then sc+1
                                  else sc)
                          p' d ps' (ins (x,y) vs) (t-1))
      TurnLeft     -> PState sc p (mod (d-1) 4) ps vs (t-1)
      TurnRight    -> PState sc p (mod (d+1) 4) ps vs (t-1)
  ins p [] = [p]
  ins p@(x,y) xs'@((x',y'):xs) = if x==x' && y==y' then xs' else
                                 if y<y' || (y==y' && x<=x')
                                 then (x,y):xs'
                                 else (x',y'):ins p xs
```

7.5 Misc. functions

```
module Misc where

import Array
import AllFudgets

amap :: (Ix a) => (b -> c) -> Array a b -> Array a c
amap f a =  array b [(i, f (a!i)) | i <- range b]
  where b = bounds a

allocGreyScale cont =
    conts (allocColorPixelF defaultColormap) rgbs
      (cont . listArray (0,white))
  where
    rgbs = [grey (i * div maxRGB white) | i<-[0 .. white]]
    white = greyLevels-1
    grey i = RGB i i i
    maxRGB = 65535 --X11 uses 16 bit color intensities

greyLevels = argReadKey "levels" 16

randomInts :: Int -> Int -> [Int]
randomInts s1 s2 = let (s1',s2') = (trunc s1,trunc s2)
  in rands s1' s2'
  where
  trunc x =
```

```
    if x < 1
    then trunc (x+2147483398)
    else if x > 2147483398
         then trunc (x-2147483398)
         else x


rands :: Int -> Int -> [Int]
rands s1 s2 = z' : rands s1'' s2''
        where   z'   = if z < 1 then z + 2147483562 else z
                z    = s1'' - s2''

                k    = s1 `quot` 53668
                s1'  = 40014 * (s1 - k * 53668) - k * 12211
                s1'' = if s1' < 0 then s1' + 2147483563 else s1'

                k'   = s2 `quot` 52774
                s2'  = 40692 * (s2 - k' * 52774) - k' * 3791
                s2'' = if s2' < 0 then s2' + 2147483399 else s2'

randomFloats :: Int -> Int -> [Float]
randomFloats s1 s2 =
  map (\x -> fromIntegral x * 4.6566130638969828e-10)
    (randomInts s1 s2)

putlistF :: (a -> [b]) -> F a b
putlistF f = mapstateF (\_ x -> ([],f x)) []
```

7.6 Polytypic functions

Note: This is not Haskell code. The function definitions must be parsed by polyp and then concatenated with the type definitions before compilation.

```
module Polyps where

add :: Int -> Int -> Int

rep :: (Int -> Int -> a -> a -> a) -> (a -> Int) ->
       Float -> Float -> a -> a -> a
rep r s m n a b = let x = (truncate . ((*) m) . fromInt . s) a
                      y = (truncate . ((*) n) . fromInt . s) b
                  in r x y a b

repl_Expr = replace_f4Expr
size_Expr = psize_f4Expr
repl_Plan = replace_f4Plan
```

```
size_Plan = psize_f4Plan
repl_Ins = replace_f0
size_Ins = psize_f0

data Plan a
     = Iffood (Plan a) (Plan a)
     | Dummy (a)
     | Move
     | TurnLeft
     | TurnRight
     | Progn2 (Plan a) (Plan a)
  deriving (Show,Read)

data Expr a
     = Bop (Char) (Expr a) (Expr a)
     | Uop (Char) (Expr a)
     | Iflte (Expr a) (Expr a) (Expr a)
     | Var (Char)
     | Num (a)
  deriving (Show,Read)

polytypic fsize :: f a Int -> Int
  = case f of
      Const t -> \x -> inc 0
      Empty   -> \x -> inc 0
      Par     -> \x -> inc 0
      Rec     -> \x -> x
      f + g   -> fsize `junc` fsize
      f * g   -> \(x,y) -> (fsize x) `add` (fsize y)

psize = cata fsize

--para :: (d a -> f a b -> b) -> d a -> b
para i x = i x (fmap id (para i) (out x))

polytypic fchildren :: (f a [b] -> [b])
  = case f of
      Const t -> \x -> []
      Empty   -> \x -> []
      Par     -> \x -> []
      Rec     -> \x -> x
      f * g   -> \(x,y) -> fchildren x ++ fchildren y
      f + g   -> fchildren `junc` fchildren

--scan :: d a -> [d a]
scan = para (\x -> \y -> x : fchildren y)
```

```
concatt xss = case xss of
               [] -> []
               (xs:xss) -> xs ++ concatt xss

replace x1 x2 p1 p2 = fst (build ((take x1 . scan) p1 ++
                          (scan . head . drop x2 . scan) p2 ++
                          (drop (x1+1) . scan) p1))

build = \(x:xs) -> let a = case xs of
                            [] -> fbuild [] (out x)
                            xs -> let b = build xs
                              in fbuild (fst b : (snd b)) (out x)
                   in (inn (fst a),snd a)

inL x = (Left (fst x),snd x)
inR x = (Right (fst x),snd x)

polytypic fbuild :: [b] -> f a b -> (f a b,[b])
  = \bs -> case f of
      Const t -> \x -> (x,bs)
      Empty -> \x -> ((),bs)
      Par   -> \x -> (x,bs)
      Rec   -> \x -> (head bs,tail bs)
      f * g -> \(x,y) -> let f1 = fbuild bs x
                             f2 = fbuild (snd f1) y
                         in ((fst f1,fst f2),snd f2)
      f + g -> \x -> case x of
                       Left x  -> inL (fbuild bs x)
                       Right x -> inR (fbuild bs x)
```

7.7 Neural network, one layer

```
module Net1 where

import Misc

data NState = NState [Float]
  deriving Show

data NInp = NInput [[Float]]
          | NLearn [Float] Float
          | NLearns [[Float]] [Float]
     deriving (Show)

type NOut = [Float]
```

```
type S a = NState -> (a,NState)

sS = NState (replicate 4 0.1)

rate :: Float
rate = 1

bias :: Float
bias = 1

c :: Float
c = 6

unitS :: a -> S a
unitS a = \s -> (a,s)

m `bindS` k = \s -> let (a,s1) = m s
                        (b,s2) = k a s1
                    in (b,s2)

sigm :: Float -> Float
sigm x = 1 / (1 + exp (-c*x))
sigm' :: Float -> Float
sigm' y = ((1/y - 1)**(1/c)) * y * y

scalp :: [Float] -> [Float] -> Float
scalp [] _ = 0
scalp _ [] = 0
scalp (x:xs) (y:ys) = x*y + scalp xs ys

errorS :: [Float] -> Float -> S [Float]
errorS is c (NState ws) = ([fx],NState (newweights is ws))
  where
  fx = sigm (scalp is ws)
  dx = rate * (c-fx) * sigm' fx
  newweights _ [] = []
  newweights (i:is') (w:ws') = w + dx * i : newweights is' ws'

errorsS :: [[Float]] -> [Float] -> S ()
errorsS iss cs (NState ws) = ((),NState (zipWith radd ws dws))
  where
  radd x y = x + rate * y / (fromInt (length iss))
  dws = addw iss cs (replicate (length ws) 0)
  addw [] [] dw = dw
  addw (is:iss') (c:cs') dw =
    addw iss' cs' (accweights is (sigm $ scalp is ws) c dw)
```

```
accweights is fx c dws = zipWith (\i w -> w + dx*i) is dws
  where
  dx = (c-fx) * sigm' fx

outputS :: [[Float]] -> S [Float]
outputS iss s@(NState ws) =
  (map (\is -> sigm (scalp (bias:is) ws)) iss,s)

neteval :: NInp -> S [Float]
neteval (NInput iss)  = outputS iss
neteval (NLearn is c) = errorS (bias:is) c
neteval (NLearns iss cs) =
  errorsS (map ((:) bias) iss) cs `bindS` \_ ->
  outputS iss
```

References

[1] *The Haskell 1.3 report*
`http://www.cs.chalmers.se/Haskell/haskell-report/`

[2] *A gentle introduction to Haskell*
`http://www.cs.chalmers.se/Haskell/haskell-report/tutorial.html`

[3] *The HBC compiler*
`http://www.cs.chalmers.se/~augustss/hbc/hbc.html`

[4] Konstantin Läufer. *Type classes and existential types*
`ftp://ftp.math.luc.edu/pub/laufer/papers/tycl-and-extypes.ps.gz`

[5] *The Fudgets system*
`http://www.cs.chalmers.se/ComputingScience/Research/Functional/Fudgets/`

[6] Patrik Jansson. *PolyP*
`http://www.cs.chalmers.se/~patrikj/poly/polyp/`

[7] Anne-Johan Annema. *Feed-forward neural networks*. Kluwer Academic publishers, 1995.

[8] John R. Koza. *Genetic programming*. MIT press, 1993.

[9] Richard Dawkins. *The blind watchmaker*. W W Norton & Co, 1988.