A Functional Approach to Sex: Reproduction in the Créatúr Framework

Amy de Buitléir[1(✉)], Mark Daly[1], Michael Russell[1], and Daniel Heffernan[2]

[1] Athlone Institute of Technology, Athlone, Ireland
amy@nualeargais.ie, {mdaly,mrussell}@ait.ie
[2] National University of Ireland, Maynooth, Ireland
dmh@thphys.nuim.ie

Abstract. Implementing genetics and reproduction for artificial life involves a set of tasks that are only loosely dependent on the type of agent or the method of reproduction. Créatúr is a software framework for automating experiments with artificial life, and a library of modules that can be used to implement agents. In this paper we describe how Créatúr uses Haskell features such as monads, domain-specific embedded languages, and datatype-generic programming to simplify the implementation of genetics and reproduction. We discuss the possibility that type families could support duplicate instance declarations in appropriate circumstances.

Keywords: Haskell · Artificial life · Datatype generic programming · Domain specific language

1 Introduction

Artificial life (ALife) is a field which attempts to create life-like behaviour using software, hardware, biochemistry or other media; in this paper we focus on software. Whereas biology is the study of "life-as-we-know-it", ALife is the study of "life-as-it-could-be"[1]. ALife is not only used as a simplified model of biological life and ecosystems; it is also increasingly applied to real-world problems as diverse as data mining[2], music composition[3], and management of dam operations in multi-reservoir river systems[4].

The recipe for evolution is simple; the ingredients are[5]:

1. *variation*: a continuing abundance of different elements,
2. *heredity or replication*: the capacity to create copies of elements, and
3. *differential fitness*: the number of copies created depends on the interaction between the features of an element with features of the environment.

All of the complexity and variation of biological life arises from this mechanism, even though "the only thing that changes in evolution is the genes"[6].

J. Hage and J. McCarthy (Eds.): TFP 2014, LNCS 8843, pp. 68–83, 2015.
DOI: 10.1007/978-3-319-14675-1_5

Although the process of evolution is normally associated with biological organisms, it can occur with any substrate as long as those three conditions are met. Hence, evolution is often used in ALife.

To explore how evolution of ALife might be implemented, consider the agent below.

```
data Plant = Plant { plantName :: String,
  plantFlowerColour :: FlowerColour, plantEnergy :: Int,
  plantGenome :: [Bool] }

data FlowerColour = Red | Orange | Yellow | Violet | Blue
```

This is of course a very simple example. There is only one genetic trait, `plantFlowerColour`; it is specified by the `plantGenome`, which is encoded as a sequence of `Bool`s. (The field `plantEnergy` is not genetic; it is set to the same initial value for all `Plant`s at "birth".)

Our `Plant` type has only *one* strand of genetic material; this illustrates a common approach[7, p. 10f] in evolutionary computation that we will refer to as *simplified sexual reproduction*. During reproduction, the strands from two parents are recombined to produce two new strands. Two offspring can be created from the new strands. Alternatively, one strand may be chosen at random to create a child, and the other strand discarded. In either case, each parent contributes approximately half of its genetic information to the offspring.

Compare the definition of `Plant` with the following definition. This agent, called `Bug`, uses an approach that more closely models sexual reproduction in biology.

```
data Bug = Bug { bugName :: String, bugColour :: BugColour,
  bugSpots :: [BugColour], bugSex :: Sex, bugEnergy :: Int,
  bugGenome :: ([Word8],[Word8]) }

data BugColour = Green | Purple | Red | Brown | Orange | Pink
  | Blue
data Sex = Male | Female
```

In this case, there are *two* strands of genetic information, represented by a tuple containing two sequences of `Word8`s. During reproduction, the two strands from one parent are recombined to produce two new strands. One of those strands is chosen at random to become that parent's contribution to the child's genome. This is analogous to the production of a *gamete* (ovum or sperm) in biology. The process is repeated for the other parent. Thus the child has two strands of genetic information, one contributed by each parent. As before, each parent contributes approximately half of its genetic information to the offspring.

Although there are differences in the details, the task of implementing either style of reproduction is very similar. The programmer must design a genome, implement recombination of genetic information, support occasional mutation of

genes, provide a means to encode a set of traits into a strand of genetic information, provide a means to decode strands of genetic information to determine the corresponding traits, and implement the construction of an agent (or solution) from the genome.

The researcher may not care about the precise design of the genome, or its implementation, only requiring that it behaves in a way that supports evolution. Specifically, the genome and the recombination technique must be designed to ensure that offspring are similar to their parents (except in the case of mutation). A straightforward conversion of numeric values to binary is not a good approach; an agent with, say, 18 legs (10010) and one with 20 legs (10100) could produce a child with 31 legs (11111) – not very similar to either parent!

So designing, implementing, and testing a genome is not trivial. Are there tools that can make this easier? As part of our research using ALife to extract knowledge from large data sets with minimal preparation or ramp-up time[8], we work with a variety of agents. We developed Créatúr[1], which is both a software framework for automating experiments with ALife and a library of modules that can be used (with or without the framework) to implement agents. We chose to implement Créatúr in Haskell based on our positive experience using it to create a neural network[9]. In this paper we describe how Créatúr uses Haskell features such as monads, domain-specific embedded languages, and datatype-generic programming to address genetics and reproduction. The full source code for Créatúr is available on GitHub[10]; a tutorial is also provided[11].

2 Datatype-Generic Programming

Generic programming is programming that references types to be specified later. The actual implementation is automatically generated when the types are finally specified. The Haskell 98 standard[12] includes some support for generic programming, in the form of derived instances, but only for six typeclasses (Eq, Ord, Enum, Bounded, Show and Read). The Glasgow Haskell Compiler (GHC) provided some extensions (Data, Typeable, Functor, Foldable, and Traversable) as part of the *Scrap Your Boilerplate* system[13–15].

GHC version 7.2 added support for *datatype-generic programming* as proposed by Magalhães et al. [16]. This lightweight and portable approach allows the programmer to specify how to derive arbitrary class instances. The key is that the "generic" type is represented at runtime using a *sum-of-products* representation, which involves the following types:

- `U1` Unit, used for constructors without arguments
- `K1` Constants, additional parameters and recursion of kind *
- `M1` Meta-information (constructor names, etc.)
- `:+:` Sum, which encodes choices between constructors
- `:*:` Product, which encodes multiple arguments to constructors

[1] *Créatúr* (pronounced kray-toor) is an Irish word meaning animal, creature, or unfortunate person.

As a result of this approach, the programmer usually only needs to write implementations for a set of base types, plus an implementation for each of the representation types above. Finally, the end user simply declares their type to be an instance of the desired type (using the `DeriveGeneric` pragma). We will show an example of this in Section 3.

3 Gene Encoding

The Créatúr library provides tools to develop an encoding scheme for a gene or an entire organism. The `Genetic` class provides the functions for encoding and decoding. Initially we defined the `Genetic` class using type families, as shown below. The function `put` writes a gene to a sequence; `get` reads the next gene in a sequence. The type `Sequence` represents an encoded gene sequence, for example, `[Bool]` or `[Word8]`.

```
class Genetic g where
  type Sequence g :: *
  put :: Sequence g -> g -> Sequence g
  get :: Sequence g -> (g, Sequence g)
```

Suppose we want to support a gene sequence type of `[Bool]`. We would create type instances of `Genetic` for each Haskell base type that we want to support, such as `Char`.

```
instance Genetic Char where type Sequence Char = [Bool] ...
```

We would also create instances for `U1`, `K1`, `M1`, `:+:` and `:*:`, as discussed in Section 2. A user will then be able to build new types using the supported base types, and declare them to be instances of `Genetic`. For example,

```
data MyType = MyType Char ... deriving Generic
instance Genetic MyType
```

A problem arises when we want to support multiple types of gene sequences for the base types. For example, we might wish to add the following:

```
instance Genetic Char where type Sequence Char = [Word8] ...
```

Note that the type signatures of `put` and `get` reference `Sequence`, so *in theory* the compiler would always be able to determine which function instance (the `[Bool]` or the `[Word8]` version) to call. However, the current implementation of type families does not permit duplicate instance declarations. One way to achieve a similar result is to create `newtype` "wrappers" for each instance declarations, as shown below.

```
newtype CharB = MkCharB Char
instance Genetic CharB where type Sequence CharB = [Bool] ...

newtype CharW8 = MkCharW8 Char
instance Genetic CharW8 where type Sequence CharW8 = [Word8] ...
```

The user can now create new types based on the "wrapped" versions of base types, and automatically derive instances for them, as shown below.

```
data MyType = MyType CharB ... deriving Generic
instance Genetic MyType
```

However, suppose the user now wants to change from using `[Bool]` for encoded gene sequences to `[Word8]`. Every reference to `CharB` will have to be changed to `CharW8`. References to other base types will have to be modified similarly. Worse still, suppose the user wants to use both `[Bool]` and `[Word8]` sequences in the same program. They would have to define multiple versions of their types. This situation is not user-friendly.

Another possibility is to use multi-parameter typeclasses, as shown below. However, we felt that multi-parameter typeclasses were less likely to be familiar to our users than type families[2].

```
class Genetic s g where
  put :: s -> g -> s
  get :: s -> (g, s)

instance Genetic [Bool] Char where ...
instance Genetic [Word8] Char where ...
```

Ultimately we chose to follow the model of commonly-used modules such as `Data.ByteString` and `Data.Map`, i.e., having multiple modules that provide the same interface. By simply changing the `import` statement, the user can change the sequence type. This makes it easy for the user to benchmark different types to determine, for example, whether `[Word8]` or `[Word16]` will be more efficient in a given application.

We also provided `Reader` and `Writer` monads for operating on an encoded gene sequence. These will be discussed in more detail in Section 7. The final implementation of `Genetic` is shown below.

[2] Hage reported that in 2010, Type Families were enabled in 114 packages on Hackage, and one of the top 10 downloads at that time, while MultiParamTypeClasses were enabled in 321 packages, and nine of the top 10[17]. However, Type Families are a more recent development than MultiParamTypeClasses, and may be overtaking MultiParamTypeClasses in popularity. On 27 June, 2014, we searched stackoverflow (http://stackoverflow.com), a Q&A forum for programmers, for the tag `[haskell]` and the term `TypeFamilies`. This search yielded 81 questions asked or answered during the year to date (409 for all time). A search for `[haskell]` and `MultiParamTypeClasses` yielded 55 questions during the year (349 for all time).

```
class Genetic g where
  put :: g -> Writer ()
  get :: Reader (Either [String] g)
```

Datatype-generic programming allows Créatúr to automatically generate instances for `put` and `get`. The details of how to use datatype-generic programming are described by Magalhães [16] and on the Haskell wiki[18]. Here we will summarise the steps we took to allow implementations of the `Genetic` class to be automatically generated.

- Implementing `Genetic` for a set of base types `Bool`, `Char`, `Word8` and `Word16`, along with types of the form `[a]`, `Maybe a`, `(a, b)` and `Either a b`, where `a` and `b` are themselves instances of `Genetic`.
- Creating a new class, `GGenetic`, which handles encoding and decoding of the sum-of-products representation of a value.
- Implementing `GGenetic` for each of the types used in the sum-of-products representation.
- Providing a default implementation of `put` and `get` in the `Genetic` class; they simply invoke the corresponding methods in the `GGenetic` class.

As a result, the end user can automatically create an instance of `Genetic` for any type without writing an implementation for `put` or `get`, as long as the type is constructed using only the supported base types. For example, we can modify the `FlowerColour` type to use the automatically-generated genetic encoding scheme by using the language pragma `DeriveGeneric`, importing `GHC.Generics`, and declaring `FlowerColour` to be an instance of `Genetic`. Now `get` and `put` can be used with the `FlowerColour` type.

```
{-# LANGUAGE DeriveGeneric #-}
...
import ALife.Creatur.Genetics.BRGCBool
import GHC.Generics
...

data FlowerColour = Red | Orange | Yellow | Violet | Blue
  deriving Generic
instance Genetic FlowerColour
```

There are three variants of `Genetic`. The one in `ALife.Creatur.Genetics.Code.BRGCBool` encodes genes to produce a sequence of `Bool`s. This is practical when the genes of an agent have a small set of possible values. If an agent has genes with a larger number of possible values, it may be better to store their genetic information as a string of numbers. `ALife.Creatur.Genetics.Code.BRGCWord8` encodes genes to produce a string of `Word8`s. Similarly, `ALife.Creatur.Genetics.Code.BRGCWord16` uses `Word16`s.

All three implementations encode integral and character values using a *binary-reflected Gray code* (BRGC). A Gray code maps values to codes in

a way that guarantees that the codes for two consecutive values will differ by only one bit[19]. This feature is useful for encoding genes because the result of a crossover operation will be similar to the inputs. This helps to ensure that offspring are similar to their parents, as any radical changes from one generation to the next are the result of mutation alone.

4 Reproduction

Recall that in our `Plant` example, each agent has a *single* strand of genetic information. During reproduction, the strands from two parents are recombined, creating genetic information for potential offspring. Thus, each parent contributes approximately half of its genetic information to the offspring. The recombination process will be discussed in Section 5.

Créatúr provides the `Reproductive` class in the `ALife.Creatur.Genetics.Reproduction.SimplifiedSexual` module for this purpose. This class can be used with either `BRGCBool`, `BRGCWord8` or `BRGCWord16`, and contains three functions. The function `recombine` recombines the genetic information from two potential parent agents, as discussed above. The user must provide the implementation for `recombine` using a domain-specific embedded language (DSEL) which will be described in Section 5. The function `build` constructs an agent from a strand of genetic information, if it is possible to do so (i.e. if the genes translate to a valid agent). The user must provide an implementation of this function as well; this is discussed in Section 7. Finally, the `makeOffspring` function takes two agents and attempts to produce offspring. A default implementation is provided, which calls `recombine` to create a genome for the child and calls `build` to construct the child. The definition of `Reproductive` is shown below.

```
class Reproductive a where
  type Strand a
  recombine :: RandomGen r => a -> a -> Rand r (Strand a)
  build :: AgentId -> Strand a -> Either [String] a
  makeOffspring :: RandomGen r
    => a -> a -> AgentId -> Rand r (Either [String] a)
```

In our `Bug` example, each agent has *two* strands of genetic information. During reproduction, the two strands from one parent are recombined to produce two new strands. (The recombination process will be discussed in Section 5.) One of these strands is chosen at random to become that parent's contribution to the child's genome. This is analogous to the production of a *gamete* (ovum or sperm) in biology. The process is repeated for the other parent. Thus the child has two strands of genetic information, one contributed by each parent. As before, each parent contributes approximately half of its genetic information to the offspring.

Créatúr provides a class for this, also called `Reproductive`, in the `ALife.Creatur.Genetics.Reproduction.Sexual` module. As before, this class can be used with either of the encoding methods described in Section 3, and contains

three functions. The `produceGamete` function recombines the twin strands of genetic information from two potential parents, using the technique described above. The user must provide the implementation for `recombine` using the DSEL described in Section 5. The function `build` constructs an agent from two strand of genetic information, if possible. The user must provide an implementation of this function; this will be discussed in Section 7.

Finally, the `makeOffspring` function takes two agents and attempts to produce offspring. A default implementation is provided, which calls `produceGamete` to produce a single strand of genetic information from each parent, pairs the two strands to create a genome for the child, and calls `build` to construct the child.

The definition of `Reproductive` is shown below.

```
class Reproductive a where
  type Strand a
  produceGamete :: RandomGen r => a -> Rand r (Strand a)
  build :: AgentId -> (Strand a, Strand a) -> Either [String] a
  makeOffspring :: RandomGen r
    => a -> a -> AgentId -> Rand r (Either [String] a)
```

5 Gene Recombination

Both of the scenarios described in Section 4 involve shuffling a pair of sequences to produce two new pairs, and possibly discarding one of the sequences. Additionally, occasional random mutations are allowed. The `ALife.Creatur.Genetics.Recombination` module in the Créatúr library provides a DSEL for genetic recombination. These operations can be applied with specified probabilities and combined in various ways. Two common operations are *crossover* and *cut-and-splice*. In crossover (Figure 1), a single crossover point is chosen. All data beyond that point is swapped between strings. In cut-and-splice (Figure 2), two points are chosen, one on each string. This generally results in two strings of unequal length.

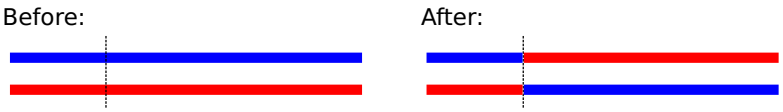

Fig. 1. Crossover

Here's a sample program that might be used to shuffle two sequences of genetic material.

```
withProbability 0.1 randomCrossover (xs, ys) >>=
withProbability 0.01 randomCutAndSplice >>=
withProbability 0.001 mutatePairedLists >>=
randomOneOfPair
```

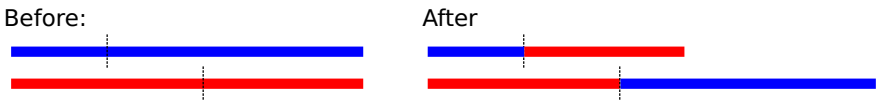

Fig. 2. Cut-and-splice

To illustrate how this program would work, suppose this program acted on the following pair of sequences:

```
([A,A,A,A,A,A,A,A,A,A],[C,C,C,C,C,C,C,C,C,C])
```

The first line of the program has a 10% probability of performing a simple crossover at a random location, perhaps resulting in:

```
([A,A,A,A,A,A,A,A,C,C,C],[C,C,C,C,C,C,C,C,A,A,A])
```

The second line of the program has a 1% probability of performing a cut-and-splice, perhaps resulting in:

```
([A,A,A,A,C,A,A,A],[C,C,C,C,C,C,C,A,A,A,C,C,C])
```

The third line of the program has a 0.1% probability of mutating one or both sequences, perhaps resulting in

```
([T,A,A,A,C,A,A,A],[C,C,C,C,C,C,C,A,A,C,C,C,C])
```

After the first three operations, we have two new sequences. In this example, we only want one of the sequences, so the final line randomly chooses one.

To perform more than one crossover, the operation can simply be repeated. Alternatively, we can choose the number of crossover operations at random. The function `repeatWithProbability` performs an operation a random number of times, such that the probability of repeating the operation `n` times is p^n. Table 1 contains the full list of available operators.

6 Gene Expression

In biology, gene *expression* is the mechanism that determines the *phenotype* (the observable traits of the organism) from the *genotype* (genetic makeup)[20]. Similarly, an ALife species which uses sexual reproduction needs a way to determine the characteristics of an agent from the two strands of genetic information. An agent that arises through sexual reproduction will have *two* strands of genetic information. When corresponding genes from the two sets are not identical, the dominance rules are applied.

The `Diploid` class, in the module `ALife.Creatur.Genetics.Diploid`, represents paired genes or paired instructions for building an agent. `Diploid` contains the function `express`. Given two possible forms of a gene or gene sequence, `express` takes into account any dominance relationship, and returns a gene representing the result. Créatúr uses *datatype-generic programming* (discussed in Section 3) to provide a default implementation of `Diploid`, including `express`. The definition of `Diploid` is shown below.

Table 1. The Recombination DSEL

function and description
`crossover :: Int -> ([a], [a]) -> ([a], [a])` Cuts the list `xs` at position `n`, cuts the list `ys` at position `m`, swaps the ends, splices them, and returns the modified pair. The result will be `(xs[0..n-1]++ys[m..], ys[0..m-1]++xs[n..])`
`cutAndSplice :: Int -> Int -> ([a], [a]) -> ([a], [a])` Cuts both the lists `xs` and `ys` at position `n`, swaps the ends, splices them, and returns the modified pair. This is equivalent to `cutAndSplice n n (xs,ys)`.
`mutateList :: (Random n, RandomGen g) => [n] -> Rand g [n]` Mutates a random element in the list `xs`, and returns the modified list.

`mutatePairedLists`
`:: (Random n, RandomGen g) => ([n], [n]) -> Rand g ([n], [n])`
Randomly chooses `xs` or `ys`, mutates a random element in that list, and returns the modified list.

`randomOneOfList :: RandomGen g => [a] -> Rand g a` Randomly returns one element from the list `xs`.
`randomOneOfPair :: RandomGen g => (a, a) -> Rand g a` Randomly returns `x` or `y`.
`randomCrossover :: RandomGen g => ([a], [a]) -> Rand g ([a], [a])` Same as `crossover`, except that `n` is chosen at random.
`randomCutAndSplice :: RandomGen g => ([a], [a]) -> Rand g ([a], [a])` Same as cutAndSplice, except that `n` and `m` are chosen at random.

`withProbability`
`:: RandomGen g => Double -> (b -> Rand g b) -> b -> Rand g b`
Either applies `op` to `x` (with probability `p`) and returns the result, or returns the unmodified `x` (with probability `p-1`).

`repeatWithProbability`
`:: RandomGen g => Double -> (b -> Rand g b) -> b -> Rand g b`
Applies `op` to `x` random number of times. The probability of applying `op` `n` times is p^n.

```
class Diploid g where
  express :: g -> g -> g
```

Default implementations of `Diploid` are provided for the following types: `Bool`, `Char`, `Double`, `Int`, `Word`, `Word8`, `Word16`, `Word32`, and `Word64`, along with sequences, tuples, and sums or products of any types that themselves implement `Genetic`. In practice, this means that the user can often create an instance of `Diploid` without writing an implementation for `express`.

In the default implementation of `express` "small" is dominant over "large". If arrays are of different lengths, the result will be as long as the shorter array.

```
express [1,2,3,4] [5,6,7,8,9]  →  [1,2,3,4]
```

Consider the following type:

```
data MyType = MyTypeA Bool | MyTypeB Int
  | MyTypeC Bool Int [MyType] deriving (Show, Generic)
instance Diploid MyType
```

Here are some examples of how `express` operates.

```
express (MyTypeA True) (MyTypeA False)  →  MyTypeA True
express (MyTypeB 2048) (MyTypeB 36)     →  MyTypeB 36
```

When a type has multiple constructors, the constructors that appear earlier in the definition are dominant over those that appear later. For example:

```
express (MyTypeA True) (MyTypeB 7)        → MyTypeA True
express (MyTypeB 4) (MyTypeC True 66 []) → MyTypeB 4
```

Even with complex data structures, the implementation should just "do the right thing".

Given a numeric type, it would seem that the logical way to express two values is to average them. So why do we instead use the smaller value? In our research with ALife, numeric genes usually control the resources used by an agent. Examples include a gene which specifies the number of neural connections in the agent's brain, or a gene which controls the age at which offspring become mature and are no longer dependent on a parent. Choosing the smaller number helps to ensure that agents use resources efficiently. Of course, a different dominance rule can be used by writing a custom implementation of `express`.

7 Constructing an Agent from Its Genome

Monads "provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism" [21]. Since a monad defines a small set of operations that can be used within it, it is essentially a DSEL. Hudak calls monads used in this way "modular monadic interpreters" because they allow different language features to be isolated and given context-specific interpretations, and combined like "building blocks" [22]. In this section we will demonstrate how we use monads to create tools for constructing agents.

As mentioned in Section 4, implementations of the class `Reproductive` must implement the function `build`, which constructs an agent from a genome, if the genome is valid. We are now ready to show how this is done.

Recall the definition of `Plant` from Section 1. To create a plant, we need to determine the flower colour from the genome, and set the ID and energy. The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules define a monad called `Reader` (unrelated to `Control.Monad.Reader`), which provides functions for decoding a

Table 2. The Reader DSEL

function and description
`get :: Reader (Either [String] g)` Reads the next gene. If it can be decoded, returns the decoded value. Otherwise, returns a list of error messages.
`getWithDefault :: g -> Reader g` Reads the next gene. If it can be decoded, returns the decoded value. Otherwise, returns the default value
`copy :: Reader Sequence` Return the entire genome.
`consumed :: Reader Sequence` Return the portion of the genome that has been read (by `get` or `getWithDefault`).

strand of genetic information. Thus, the `Reader` monad is a DSEL for reading genomes; this language is defined in Table 2.

We can write a `buildPlant` method using this DSEL. The function will take a `String` (a unique identifier of the plant to be created), and it will return a program that runs in the `Reader` monad. That program will return a either a list of `String`s containing error messages, or a plant. Thus, the type signature for the `buildPlant` function is:

```
buildPlant :: String -> Reader (Either [String] Plant)
```

Now to write the program. First, each plant needs a copy of its genome in order to produce offspring; we can use the `copy` function to obtain this. Next, we determine the colour of the plant. We could use the method `get`, which returns a `Maybe` value containing the next gene in a sequence. But consider that our sequence of `Bool`s may not be a valid code for any colour. If an error occurs, we could treat the mutation as non-viable and return `Nothing`. However, in this example, we wish to create a plant no matter what errors are in the genome, so we will use `getWithDefault`, with `Red` as the default value. All plants start life with an energy of `10`. Here is the program:

```
buildPlant name = do
  g <- copy
  colour <- getWithDefault Red
  return . Right $ Plant name colour 10 g
```

Now, `buildPlant` is a function that returns a program that runs in the `Reader` monad. How do we run that program? `ALife.Creatur.Genetics.BRGCBool`, `ALife.Creatur.Genetics.BRGCWord8` and `ALife.Creatur.Genetics.BRGCWord16` provide a function for this purpose, called `runReader`. Now we have everything we need to declare `Plant` to be an instance of `Reproductive`.

Table 3. The DiploidReader DSEL

function and description
`getAndExpress :: (Genetic g, Diploid g) => DiploidReader (Either [String] g)` Reads the next pair of genes from twin strands of genetic information. If the genome can be decoded, takes into account any dominance relationship and returns returns the decoded value. Otherwise, returns a list of error messages.
`getAndExpressWithDefault :: (Genetic g, Diploid g) => g -> DiploidReader g` Reads the next pair of genes from twin strands of genetic information. If the genome can be decoded, takes into account any dominance relationship and returns returns the decoded value. Otherwise, returns the default value
`copy2 :: DiploidReader DiploidSequence` Returns the entire genome (both strands).
`consumed2 :: DiploidReader DiploidSequence` Returns the portion of each strand that has been read (by `get` or `getWithDefault`).

```
instance Reproductive Plant where
  type Base Plant = Sequence
  recombine a b =
    withProbability 0.1
        randomCrossover (plantGenome a, plantGenome b) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runReader (buildPlant name)
```

Recall the definition of `Bug` from Section 1. Now we have two strands of genetic information which determine the bug's traits. The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules define a monad called `DiploidReader` for this situation. The `DiploidReader` monad is also DSEL; this language is defined in Table 3.

Our `buildBug` method will take a `String` (a unique identifier), and it will return a program that runs in the `DiploidReader` monad. The implementation is similar to `buildPlant`, except that the single-strand operations have been replaced with versions that work with both strands.

```
buildBug :: String -> DiploidReader (Either [String] Bug) buildBug
name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- copy2
  return
    $ Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g
```

The `runDiploidReader` function runs a program written in the `DiploidReader` DSEL and returns the result. Now we can implement `Reproductive`.

```
instance Reproductive Bug where
  type Base Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug False name)
```

The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules also define a monad called `Writer`, used for encoding genetic information. This is useful for generating an initial population. The `Writer` DSEL consists of one function, `put`, which writes a gene to a sequence.

One approach to creating an initial population is to feed random strings of genetic information into the function that builds the agent, but instruct it to keep only as much of the sequence as it needs to build a complete agent. The functions `consumed` (from the Reader DSEL) and `consumed2` (from the DiploidReader DSEL) are useful here. For example, we can modify the `buildBug` method from Section 7 to accept a boolean that tells it whether or not to discard the unread portion of the sequences.

```
buildBug :: Bool -> String -> DiploidReader (Either [String] Bug)
buildBug truncateGenome name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- if truncateGenome then consumed2 else copy2
  return $
    Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g
```

8 Conclusion

As library developers, we found it straightforward to use the datatype-generic programming feature of GHC to specify how to derive instances of `Genetic` and `Diploid`. The user has it even easier; they can simply declare their custom types to be instances of these classes, taking advantage of the default implementation we provide. Perhaps type families could support duplicate instance declarations in appropriate circumstances, or alternatively, type families and multi-parameter typeclasses were unified into one language feature with a common design.

Each of the DSELs we developed required only a small set of operations; it was easy to embed them in Haskell. In this way we avoided having to design

a language and write a parser for it. The user does not have to learn a "new" language, and rather than being restricted to the semantics of the DSEL, the user has access to all the features of Haskell, if needed. Finally, using monads for the `Reader`, `DiploidReader` and `Writer` DSELs allowed us to isolate the stateful computations required to read and write genes.

References

1. Langton, C.G.: Artificial life. In: Langton, C.G. (ed.) Artificial Life: The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, Held in September, 1987, in Los Alamos, New Mexico, pp. 1–48. Addison-Wesley, Redwood City (1989)
2. Cao, L.: Data Mining and Multi-agent Integration. Springer, Boston (2009). http://dx.doi.org/10.1007/978-1-4419-0522-2
3. Miranda, E.R.: A-life for music: music and computer models of living systems. A-R Editions, Middleton (2011). http://www.worldcat.org/search?qt=worldcat_org_all&q=9780895796738
4. Dessalegne, T., Nicklow, J.: Artificial Life Algorithm for Management of Multi-reservoir River Systems. Water Resources Management **26**(5), 1125–1141 (2012). http://dx.doi.org/10.1007/s11269-011-9950-7
5. Dennett, D.C.: Consciousness explained. Penguin (1993). http://www.worldcat.org/isbn/9780140128673
6. Fuller, S., Wolpert, L.: Transcript of the debate between Professor Steve Fuller and Professor Lewis Wolpert at Royal Holloway College (February 2007). http://www.bcseweb.org.uk/index.php/Main/RoyalHollowayCollegeDebate (cited July 03, 2010, 01:11:46)
7. Mitchell, M.: An introduction to genetic algorithms, 2nd edn. Prentice Hall of India, New Delhi (2002)
8. de Buitléir, A., Russell, M., Daly, M.: Wains: A pattern-seeking artificial life species. Artificial Life **18**(4), 399–423 (2012)
9. de Buitléir, A., Russell, M., Daly, M.: A Functional Approach to Neural Networks. The MonadReader **21**, 5–24 (2013). http://themonadreader.files.wordpress.com/2013/03/issue21.pdf
10. de Buitléir, A.: Créatúr GitHub. GitHub repository (2014).https://github.com/mhwombat/creatur
11. de Buitléir, A.: Créatúr Tutorial (2014). https://github.com/mhwombat/creatur-examples/raw/master/Tutorial.pdf
12. Peyton-Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press, Cambridge (2003). http://www.worldcat.org/isbn/9780521826143
13. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Shao, Z., Lee, P. (eds.) In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2003, pp. 26–37. ACM, New York (2003). http://doi.acm.org/10.1145/604174.604179
14. Lämmel, R., Jones, S.P.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, pp. 244–255. ACM, New York (2004). http://doi.acm.org/10.1145/1016850.1016883

15. Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, pp. 204–215. ACM, New York (2005). http://doi.acm.org/10.1145/1086365.1086391
16. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell 2010, pp. 37–48. ACM, New York (2010). http://doi.acm.org/10.1145/1863523.1863529
17. Hage, J.: DOMain Specific Type Error Diagnosis (DOMSTED). Technical Report UU-CS-2014-019 (2014). http://www.computerscience.nl/research/techreps/repo/CS-2014/2014-019.pdf
18. contributors W. GHC. Generics (2013). Wiki page. http://www.haskell.org/haskellwiki/GHC.Generics
19. Gray, F.: Pulse code communication. Google Patents. US Patent 2,632,058 (1953). http://www.google.com/patents/US2632058
20. Lewontin, R.: The genotype/phenotype distinction. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (Summer 2011)
21. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995). http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf
22. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys (CSUR) 28(4es), 196 (1996)