An Algebraic Description of XCS

David Pätzel
University of Augsburg
Organic Computing Group
Augsburg, Germany
david.paetzel@informatik.uni-augsburg.de

Jörg Hähner
University of Augsburg
Organic Computing Group
Augsburg, Germany
joerg.haehner@informatik.uni-augsburg.de

ABSTRACT

XCS and its derivatives are some of the most prominent Learning Classifier Systems (LCSs). Since XCS's design was done "algorithm first", there existed no formal basis at its inception. Over the past 20 years, several publications analysed parts of the system theoretically but the only approach to a more holistic theory of LCSs in general was never fully adapted. We present an *algebraic* formalisation of XCS that facilitates formal reasoning about the system and serves as a complement to earlier algorithmic descriptions. Our work is meant to give a fresh impetus to XCS and LCS theory. Since we use the programming language Haskell for our formal expressions we also describe a new abstract XCS framework in the process.

CCS CONCEPTS

• **Computing methodologies → Machine learning algorithms**;
• **Theory of computation → Design and analysis of algorithms**;

KEYWORDS

Extended Classifier System, Learning Classifier System, Formalisation, Algebra, Functional Programming

ACM Reference Format:
David Pätzel and Jörg Hähner. 2018. An Algebraic Description of XCS. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208248

1 INTRODUCTION

LCSs combine evolutionary algorithms and reinforcement learning [2]. One of the most prominent of these algorithms is XCS, which was originally devised by Wilson in 1995 [23]. In spite of two decades of XCS-related research, there are only few theoretical findings about its workings—many publications on the matter either use vast simplifications or only regard parts of the system. Additionally, most of their results have only been validated through experiments instead of through formal proofs. The main reason for this might be that XCS lacked a formal basis right from the beginning; the description of the system which nearly all further research built on was algorithmic [7], which makes formal reasoning difficult [13].

One notable exception is the work by Drugowitsch and Barry [9, 10] who approached LCSs in a manner similar to how other reinforcement learning algorithms like Q-learning were designed. While the rest of the research community went about XCS in a *top-down* way (given Wilson's XCS algorithm, deduce which problems can be learned and which modifications to the algorithm improve learning etc.), they developed a theory from the *bottom up* (given a reinforcement learning problem, what properties do classifiers have and how should they be combined by an LCS). This way they provided a profound formal basis for LCSs in general from which they were also able to deduce properties of and improvements for XCS. However, while the work of Drugowitsch and Barry has been cited several times in different publications regarding XCS, their general results have not been directly built upon any further in the XCS context (albeit they were e. g. in the UCS setting [11]); reasons for this are probably their conceptual separation from other publications on XCS and their, in comparison to other XCS-related work, higher mathematical complexity. Nevertheless, since the results of Drugowitsch and Barry are definitely valuable for formal reasoning about XCS, we think that they need to be integrated more tightly into the body of XCS research in the long run.

In the present paper, we use the functional programming language Haskell (Section 2) to provide a new *top-down* formalisation of XCS that is based on *algebraic* instead of algorithmic descriptions (Section 3). We accurately model XCS's different parts and their interdependencies in a formal and mathematically pure way. Our work facilitates formal reasoning about XCS in the future and is meant to reignite the theoretic discourse. The long term vision is to be able to perform formal proofs about which kinds of problems XCS is capable of learning and which parameters should be used to do so as well as to close the gap to the work of Drugowitsch and Barry.

2 FORMALISATION USING HASKELL

The algebraic formalisation in this paper is carried out using the functional programming language Haskell [18]. While the formal definition of that language is deliberately independent of any implementation, the dialect defined by the GHC (Glasgow Haskell Compiler) [8] has become the de-facto standard and is used here as well. For some of the more abstract expressions, a number of often-used language extensions included in the GHC is needed.

The main requirement for the formalisation is that it has to be strictly mathematical. This is the most important difference to XCS's earlier algorithmic description [7] since a purely mathematical description can be reasoned about far more easily. In general, formal reasoning about algorithms is often only possible with significant overhead such as Hoare logic [13]; for XCS this is even less suitable as it is a complex algorithm making use of randomness and revolving around a stateful collection of objects [7]. By making an intermediate step towards a formal mathematical description, the effort can be reduced.

In addition, a better abstract understanding of the different parts of the system is accomplished: While XCS is usually described with fixed types of observations, conditions and actions (e. g. bit strings for observations and actions, ternary strings for conditions), our formalisation deliberately stays on an abstract level that does not dictate the structure of these primitives. This is done using type variables and classes instead of concrete types wherever possible (and sensible). In an upcoming work we plan to show how different interfaces (e. g. the original bit strings or XCSF) can be expressed using our framework.

Haskell is preferred over the usual mathematical notation out of several reasons:

- Valid Haskell code is *type safe* [18] meaning that as long as the expressions of this paper can be compiled, they are at least correct regarding their type—reducing the amount of possible mistakes.
- Haskell syntax is very similar to conventional mathematical notation anyway.
- A usable implementation of the formalised concepts is provided "automatically".

The last argument is stressed further by the fact that Haskell supports *literate programming* [15]. This technique was made use of when writing this paper, which means that its source code can be compiled to a LaTeX document as well as to a working Haskell library that can be tried out and experimented with. We aim to publish that library in the future.

For the ones not proficient in Haskell a few differences between Haskell's and conventional mathematical notation need to be pointed out (for more details, consider the standard [18] or a Haskell textbook [20]).

- $x :: t$ means that the value x has the type t.
- *Function application*: $f\ \ x$ corresponds to $f(x)$, $f'\ \ x\ \ y\ \ z$ corresponds to $f(x, y, z)$.
- There are functions with a similar syntax at the type level as well (called *type constructors*).
- Haskell supports polymorphism via a concept called *type classes*. Such a class defines which functions are necessary for a type to be part of it.
- *Context expressions* such as "$(T\ \ m) \Rightarrow \ldots$" may be read as "given a type m that is in type class $T, \ldots$".
- $f \circ g\ \$\ x$ corresponds to $(f \circ g)(x)$, the <\$> operator is similar (\$ is a function of type $(a \rightarrow b) \rightarrow a \rightarrow b$ with low operator precedence, <\$> has type $\mathit{Functor}\ f \Rightarrow (a \rightarrow b) \rightarrow f\ \ a \rightarrow f\ \ b$).

An important concept used in many functional programming languages which is vital to this work is that of a *monad*. Monads are a clean mathematical means to deal with the non-mathematical parts of a formalisation (e. g. parts that can not be "controlled"—i. e., themselves formalised—such as external systems). An in-depth discussion of monads is beyond the scope of this paper; instead, the next paragraphs try to provide the reader with an intuitive but sufficient understanding.

A monad is essentially a type constructor and several functions with certain properties [22]. Given a monad with type constructor m, a *monadic value* of type $m\ \ t$, can be thought of as a *computation* that results in a value of type t; until the computation is evaluated, the underlying value of type t only makes sense *in the context provided by m*.

An example for one kind of monad featured in this work are *random monads* [24]. Functions have the property to, given the same inputs, always return the same outputs; therefore, no function returning random values exists. However, a function of type

$(\mathit{MonadRandom}\ m) \Rightarrow \mathit{getRandom} :: m\ \mathit{Real}$

is mathematically pure: instead of returning a value, it returns a monadic value—a computation that returns a real number. The type class *MonadRandom* abstracts from the actual implementation of the random monad (e. g. from the choice of random generator implementation).

To evaluate a monadic value, the underlying computation has to be run. For this, functions like

$\mathit{evalRand} :: \mathit{Rand}\ g\ a \rightarrow g \rightarrow a$

exist (g is the type of a random generator, $\mathit{Rand}\ \ g$ is the type constructor of a certain random monad).

Monadic values from the same monad can be combined in a natural way. Therefore, the aforementioned instantiation of actual values is usually done exactly once, namely at the upmost level directly below the *main* function of a Haskell application.

3 ALGEBRAIC DESCRIPTION OF XCS

This section presents our algebraic formalisation of XCS; at that, we cover both the versions for single- as well as for multi-step problems (we call them *single-* and *multi-step XCS* respectively). Both these algorithms were devised by Butz and Wilson in their algorithmic description of XCS [7]. As we closely follow their work semantically, we will often only show the signatures of functions—their implementation can be deduced directly from the involved types and the corresponding algorithmic description.

Large parts of the formalisation are relevant to single- as well as multi-step XCS; in that case we will simply talk about *XCS*. The phrase *an XCS* always refers to a certain instance of an extended classifier system whereas the term *XCS* (without article) refers to the XCS algorithm as a whole.

3.1 Abstract system monad

An XCS interacts with a system external to it (in the following simply called "the system") by repeatedly receiving a description of that system's current state and, based on that, initiating actions to be carried out by it. After each executed action, the XCS gets assigned a reward, which could for example be based on the difference in system performance gained through that action. Butz and Wilson proposed that this reward is issued by a reinforcement component which evaluates the system's performance [7]; however, from the XCS's perspective, it might as well come directly from the system itself. While it may not always be reasonable to assume that the system is able to determine its own performance, this is a trivial abstraction since the reinforcement component evaluating the system's performance has to be interconnected tightly with it to do so.

Another capability of the system is to signal whether the *end of problem (EOP)* is reached. Strictly speaking, this mechanism is only necessary for finite multi-step problems (in the last step, an additional update is performed). However, a system may also simply constantly return *False* here. The EOP can be used to signal when a learning phase ends as well.

We now can define a type class for the types of systems[1]:

```
class (Monad m) => MonadSystem o a m where
  observe :: m o
  act :: a -> m Reward
  eop :: m Bool
```

For two given types o and a, the type m is a system if the three functions *observe*, *act* and *eop* are implemented for it. At that, m is required to be a monad (the *system monad*) while o and a are type variables that can be interpreted as the types for observations and actions respectively. The system monad abstracts away the internal state of the system which is not under the control of the XCS. It works similarly to the random monad introduced in Section 2; instead of providing a context of randomness, it provides the context of the external system's current state.

The previous definition shows the level of abstraction of our formalisation: neither the actual structure of actions, observations nor, as can be seen in the next section, of conditions is fixed. Instead, only the operations these types have to support are specified.

3.2 Conditions, actions, detectors and mutators

For a type c to be usable as the type for *conditions* its values have to be comparable (the combination of the type classes *Eq* and *Ord* require an equality relation as well as a partial order to be defined on c). The *generalises* function relates two conditions c_1 and c_2 in terms of whether c_1 matches all observations that c_2 matches. $crossover_1$ performs a single-point crossover on two conditions, returning two new conditions. Since most implementations will want to choose the crossover position at random, the resulting pair of conditions is wrapped into the random monad. Given a function for single-point crossover, a function for n-point crossover can be derived through recursion and a monad combinator.

```
class (Eq c, Ord c) => Condition c where
  generalises :: c -> c -> Bool
  crossover_1 :: (MonadRandom m) => c -> c -> m (c, c)
  crossover_n :: (MonadRandom m) => c -> c -> m (c, c)
  crossover_n c_1 c_2
    | n <= 0 = return (c_1, c_2)
    | otherwise =
      crossover_1 c_1 c_2 >>= uncurry (crossover_{n-1})
```

The *Detector* type class defines the relationship between the types for conditions and observations: An XCS matches incoming observations of type o against conditions of type c; therefore a function *matchedBy* is required. In order to support covering, a $coverCondition_{p_\#}$ function must be defined that relates a probability $p_\#$ for a higher generality (e. g. using a wildcard in a bit string condition) and an observation of type o to a random condition of type c. Covering usually requires randomness thus the random monad context.

```
class (Condition c) => Detector o c where
  matchedBy :: o -> c -> Bool
  coverCondition_{p_#} :: (MonadRandom m) => o -> m c
```

Just like types for conditions, types for *actions* are required to be comparable. The only other requirement is a function called *otherThan* that is used for covering: Given a list of actions, it either returns an action not in that list wrapped in a *Just* or *Nothing* (if the list of actions contains all possible actions, no other action can be returned). This is expressed using the *Maybe* type constructor which augments a type with the possibility to not have a value. If multiple allowed actions are not in the list, an implementation of *otherThan* may use randomness to select one.

```
class (Eq a, Ord a) => Action a where
  otherThan :: (MonadRandom m) => [a] -> m (Maybe a)
```

A *rule* consists of a condition and an action; this simplifies notation in many cases. Since conditions and actions are comparable, so is a rule.

```
data Rule c a = Rule
  {condition :: c
  , action    :: a}
  deriving (Eq, Ord)
```

To support the genetic mutation operation, any used combination of condition and action types is required to implement a $mutate_\mu$ function which is specified by the *Mutator* type class. Given a probability μ for a gene's mutation and an observation of type o that the result needs to conform to (mutation never makes a classifier leave the current niche), this function returns a rule with a mutated condition and action.

```
class (Condition c, Action a) => Mutator o c a where
  mutate_μ :: (MonadRandom m) =>
    o -> Rule c a -> m (Rule c a)
```

3.3 Classifiers

A *classifier* consists of a rule and data about its validity. That data is called the classifier's *metadata*:

```
data Metadata = Metadata
  {prediction   :: Prediction, error       :: Error
  , fitness     :: Fitness   , experience  :: Experience
  , timeStepGA  :: Integer   , avgASetSize :: Real
  , numerosity  :: Positive}
```

The type of a classifier for a condition type c and an action type a is therefore:

```
data Classifier c a = Classifier
  {rule :: Rule c a
  , md  :: Metadata}
```

[1]Slightly simplified, functional dependencies [12] are required for this expression to pass the type checker.

Two classifiers are considered equal if and only if their rules are equal.

$$\begin{array}{l}\textbf{instance}\ (Eq\ c, Eq\ a) \Rightarrow Eq\ (Classifier\ c\ a)\ \textbf{where}\\ \quad cl_1 \equiv cl_2 = rule\ cl_1 \equiv rule\ cl_2\end{array}$$

In order to shorten some of the definitions and more closely resemble the original notation, access to a classifier *cl*'s metadata fields is possible via $g\ cl$ where g is one of the accessor functions devised by Butz and Wilson [7]:

$$g \in [a, c, p, \epsilon, f, exp, ts, as, num]$$

For example, the classifier's fitness can be accessed as usual via $f\ \ cl$ instead of $fitness \circ md\ \$\ cl$.

For the sake of brevity, most of the functions defined on classifiers are left out here and only the function *subsumes* is shown as an example. It closely follows the definition of the `DOES SUBSUME` procedure by Butz and Wilson [7]. The value of type *Config* contains the XCS's parameters—in this case, the function *couldSubsume* uses the parameters θ_{sub} and ϵ_0 to assess whether cl_1 is experienced and accurate enough to subsume another classifier.

$$\begin{array}{l}subsumes :: (Condition\ c, Eq\ a) \Rightarrow\\ \quad Config \longrightarrow Classifier\ c\ a \longrightarrow Classifier\ c\ a \longrightarrow Bool\\ subsumes\ conf\ cl_1\ cl_2 =\\ \quad a\ cl_1 \equiv a\ cl_2 \land\\ \qquad couldSubsume\ conf\ cl_1 \land\\ \qquad c\ cl_1\ \text{`}generalises\text{`}\ c\ cl_2\end{array}$$

$$\begin{array}{l}couldSubsume :: Config \longrightarrow Classifier\ c\ a \longrightarrow Bool\\ couldSubsume\ conf\ cl = exp\ cl > \theta_{sub}\ conf \land \epsilon\ cl < \epsilon_0\ conf\end{array}$$

3.4 Population

The XCS keeps a set of classifiers in a structure called the *population* for which several possible interfaces exist; the one presented here strikes a decent balance between the number of supported operations and their generality. It is specified by a type class: Given types for conditions and actions, *c* and *a*, a type $p\ c\ a$ is a population if the following functions are defined for it.

$$\begin{array}{l}\textbf{class}\ Population\ p\ c\ a\ \textbf{where}\\ \quad empty :: p\ c\ a\\ \quad size :: p\ c\ a \longrightarrow NonNegative\\ \quad retrieve :: Rule\ c\ a \longrightarrow p\ c\ a \longrightarrow Maybe\ (Classifier\ c\ a)\\ \quad matching :: (Detector\ o\ c) \Rightarrow o \longrightarrow p\ c\ a \longrightarrow [Classifier\ c\ a]\\ \quad insertWith ::\\ \qquad (Metadata \longrightarrow Metadata \longrightarrow Metadata) \longrightarrow\\ \qquad\quad Classifier\ c\ a \longrightarrow p\ c\ a \longrightarrow p\ c\ a\\ \quad update :: (Metadata \longrightarrow Maybe\ Metadata) \longrightarrow\\ \qquad Rule\ c\ a \longrightarrow p\ c\ a \longrightarrow p\ c\ a\\ \quad foldr :: (Classifier\ c\ a \longrightarrow y \longrightarrow y) \longrightarrow y \longrightarrow p\ c\ a \longrightarrow y\end{array}$$

At that, these semantics must hold (only listing the non-obvious ones):

- If a classifier *cl* with rule *l* exists in population *p*, $retrieve\ l\ p$ is $Just\ cl$ (otherwise it is *Nothing*).
- $matching\ \ o\ \ p$ is a list of classifiers in *p* whose conditions match *o*.
- $insertWith\ f\ \ cl\ \ p$ equals *p* but with the classifier cl_1 inserted. If a classifier cl_2 with rule $rule\ cl_1$ already exists in *p*, that classifier's *metadata* is updated to be $f\ (md\ cl_2)\ (md\ cl_1)$.
- $update\ f\ \ l\ p$ updates the classifier with rule *l* in *p* using *f*. If *f* returns *Nothing*, the classifier is deleted.
- *foldr* uses the given right-associative operator to fold the population to a single value.

While all the functions defined here could be expressed using *foldr* alone, for many underlying collection types this would result in a non-optimal performance (e. g. if a hash map were used, *retrieve* could be $O(1)$ instead of *foldr*'s $O(n)$).

Note that the population is *not* updated in place as is the usual case in imperative languages; instead, an updated copy of the population is returned. This, again, is due to the purely functional paradigm this work follows.

3.5 State in the XCS

In most publications about XCS, the XCS manages the population by updating it in place. The same applies to the step counter, which is used, for example, to determine whether an iteration of the *genetic algorithm* (GA) should be performed. Some XCSs perform parameter optimisation or adjustment during runtime (e. g. adapting the learning rate); thus it makes sense to consider the configuration to be updateable as well. All in all, the state managed by an XCS can be expressed like so:

$$\begin{array}{l}\textbf{data}\ State\ p\ c\ a = State\\ \quad \{config \quad :: Config\\ \quad ,population :: p\ c\ a\\ \quad ,time \quad\ \ :: TimeStep\}\end{array}$$

The upcoming sections make use of this data structure: they retrieve configuration values, update classifiers from the population as well as increase the current time step. In imperative programming languages these operations could be carried out by reading and writing a global variable containing the current state. In the setting of this paper, however, this is not possible: there are no mutable values like that in pure mathematics. To cope with that, we utilise *state monads*, whose mechanics are briefly introduced by example next.

Assume that there are two *procedures* with the following signatures.

$$f :: a \longrightarrow b; g :: b \longrightarrow c$$

Further assume that they are purely functional save for *reading* and *writing* the value of an object of type *S* (the *state object*). Then, these procedures can be "transformed" into pure functions by adding a parameter of that type, yielding:

$$\begin{array}{l}f' :: S \longrightarrow a \longrightarrow (b, S)\\ g' :: S \longrightarrow b \longrightarrow (c, S)\end{array}$$

Now each of these functions has read access to a state object: they are provided with one via a parameter. In addition, they each return a pair which contains the new, updated state object. Programming with these functions now consists of applying them to the current state and retrieving the new state for the next function call:

$$(g' \circ f')\ s\ a = \textbf{let}\ (b, s') = f'\ \ s\ a\ \textbf{in}\ g'\ \ s'\ \ b$$

Here, a **let**-expression was used to bind the values of the tuple returned by f'.

This whole mechanism can be made abstract by using a state monad [17, 22], which does nothing more than "automatically" adding the corresponding parameters to functions. General state monad–versions of f' and g' are:

$f'' :: (\mathit{MonadState}\ S\ m) \Rightarrow a \rightarrow m\ b$
$g'' :: (\mathit{MonadState}\ S\ m) \Rightarrow b \rightarrow m\ c$

With that, the composition of f'' and g'' can be written as

$(g'' \circ f'')\ a = f''\ a \ggg g'' :: (\mathit{MonadState}\ S\ m) \Rightarrow m\ c$

and the final value of type c can be retrieved using *evalState* which provides the function chain with an initial state value s of type S.

$\mathit{evalState}\ (f''\ a \ggg g'')\ s :: c$

Again, the above-introduced mental model of monadic values such as $f''\ a$ being composable computations is helpful here.

A state monad is used in the following definitions whenever the "current" value of the population, step counter or configuration has to be retrieved or updated. An example is the function with signature

$\mathit{increaseTime} :: (\mathit{MonadState}\ (\mathit{State}\ p\ c\ a)\ m) \Rightarrow x \rightarrow m\ x$

which may be read as if *increaseTime* received an additional parameter of type $\mathit{State}\ p\ c\ a$ and returned a pair of type $(x, \mathit{State}\ p\ c\ a)$. The function increases the tick counter by one (thus the use of the state monad's *modify* function); for a better composability, it receives one argument of an arbitrary type x and returns it without modification.

$\mathit{increaseTime}\ x =$
$\quad \mathit{modify}\ (\lambda s \rightarrow s\ \{\mathit{time} = \mathit{time}\ s + 1\}) \gg \mathit{return}\ x$

3.6 Matching and covering

Whenever the XCS receives an observation, the first task is to find all existing classifiers whose conditions match. If the classifiers in that set propose less than θ_{MNA} different actions, covering occurs, which means that new matching classifiers are created randomly until this condition is met. Since the XCS as specified by Butz and Wilson starts with an empty population, $\theta_{MNA} = 0$ leads to no classifiers being created at all; thus $\theta_{MNA} > 0$ can be assumed. Therefore the final set of matching classifiers (the *match set*) can never be empty, which can be expressed at the type level by a non-empty list.

type $\mathit{MSet}\ c\ a = \mathit{NonEmpty}\ (\mathit{Classifier}\ c\ a)$

The classifiers created by covering are required to match the current observation; thus that observation is a parameter of the *cover* function. In order to only need to perform the expensive matching operation on the whole population once, already existing matching classifiers are provided to *cover* as well which recurses until that value fulfils the θ_{MNA} requirement. Retrieving the configuration value θ_{MNA} as well as inserting classifiers into the population means utilising the state monad introduced in the previous section. Since the creation of new classifiers makes use of the *otherThan* function of *Action*, the random monad context is required as well. The *Detector* relationship between the types of the observation and the condition provides the $\mathit{coverCondition}_{p_\#}$ function. The function signature of *cover* is therefore:

$\mathit{cover} ::$
$\quad (\mathit{Population}\ p\ c\ a, \mathit{Detector}\ o\ c, \mathit{Action}\ a,$
$\qquad \mathit{MonadRandom}\ m, \mathit{MonadState}\ (\mathit{State}\ p\ c\ a)\ m) \Rightarrow$
$\quad o \rightarrow [\mathit{Classifier}\ c\ a] \rightarrow m\ (\mathit{MSet}\ c\ a)$

The *matchSet* function now simply maps observations to match sets. Its concise implementation makes use of two combinators known from functors and monads:

$\mathit{matchSet} ::$
$\quad (\mathit{Population}\ p\ c\ a, \mathit{Detector}\ o\ c, \mathit{Action}\ a,$
$\qquad \mathit{MonadRandom}\ m, \mathit{MonadState}\ (\mathit{State}\ p\ c\ a)\ m) \Rightarrow$
$\quad o \rightarrow m\ (\mathit{MSet}\ c\ a)$
$\mathit{matchSet}\ o = (\mathit{matching}\ o <\$> \mathit{gets}\ \mathit{population}) \ggg \mathit{cover}\ o$

3.7 Action selection

The next step is to select one of the actions proposed by the match set. First, the classifiers in the match set are grouped according to their proposed action and the *system prediction* of these groups is evaluated. This results in the following data structure called the *prediction array*. Note that contrary to the prediction array devised by Butz and Wilson [7] the classifiers themselves *are* a part of it. Actions that were not proposed make no occurrence.

type $\mathit{PArray}\ c\ a =$
$\quad \mathit{NonEmpty}$
$\qquad (a, (\mathit{SystemPrediction}, \mathit{NonEmpty}\ (\mathit{Classifier}\ c\ a)))$

By selecting one of the proposed actions from the prediction array an *action set* is formed. The original work [7] devised this as a simple collection of classifiers; however, its semantics entail more than that.

There are several different strategies to select actions; the one chosen by Butz and Wilson [7] is the ϵ-greedy selection which performs exploration with a certain probability p_{explr} and exploitation otherwise. Since later on the handling of an action set will differ based on whether it was formed during an explore or an exploit step, it gets annotated with a *mode* (other selection strategies distinguish between these two modes as well, especially in online learning settings).

data $\mathit{Mode} = \mathit{Exploring} \mid \mathit{Exploiting}$

The type of an action set formed through action selection given a prediction array of type $\mathit{PArray}\ c\ a$ can then be expressed as follows. Again, the underlying set of classifiers is guaranteed to be non-empty—the action simply could not have been chosen if no classifier proposed it.

data $\mathit{ASet}\ c\ a = \mathit{ASet}$
$\quad \{\mathit{mode} \qquad :: \mathit{Mode}$
$\quad , \mathit{proposing} :: \mathit{NonEmpty}\ (\mathit{Classifier}\ c\ a)\}$

Accordingly, the signature of the ϵ-greedy selection function is:

$\mathit{selectEpsilonGreedy}_{p_{explr}} :: (\mathit{Action}\ a, \mathit{MonadRandom}\ r) \Rightarrow$
$\quad \mathit{PArray}\ c\ a \rightarrow r\ (\mathit{ASet}\ c\ a)$

3.8 XCS for single-step problems

In order to be able to succinctly describe XCS's main loop, we first specify the type for an XCS computation. By employing monad transformers this can be done in a modular and abstract way [17]. The following type definition can be interpreted as: Given types for a system monad (*io*), a random generator (*g*), a population (*p*), observations (*o*), conditions (*c*) and actions (*a*), an XCS computation is a computation dependent on the external system augmented with randomness-dependence and the state monad managing a *State p c a*.

```
newtype XCS io g p o c a x = XCS
  { runXCS′ :: StateT (State p c a) (RandT g io) x }
```

The *interaction* function returns an XCS computation that performs one XCS step save for the classifier update. Since single- and multi-step XCS's only differ in that update, this function is utilised by both. Note that the **do**-notation is mere syntactic sugar; while it closely resembles imperative programming, its semantics are different. Each statement in this **do**-block must be an XCS computation or a **let**-expression and each line break can be desugared to a combination of a certain monad combinator (>>= or >>) and a lambda expression [18].

```
interact ::
  (P.Population p c a, Detector o c, Mutator o c a,
    RandomGen g, MonadSystem o a io) ⇒
  XCS io g p o c a (Result o c a)
interact = do
  p_explr ← C.p_explr <$> gets config
  o ← observe
  mset ← matchSet o
  let parray = predictionArray mset
  aset ← selectEpsilonGreedy_{p_explr} parray
  r ← act ∘ proposition $ aset
  let pSystem = maximum (fst ∘ snd <$> parray)
  return $ Result o pSystem aset r
```

The *interact* function returns a value of type *Result* which is a simple sum type introduced to increase descriptiveness of function signatures in the following; it contains all the values required for a single- or multi-step update.

```
data Result o c a =
  Result o SystemPrediction (ASet c a) Reward
```

In a single-step XCS, the classifiers are always updated only regarding information about the current step; thus this operation depends on a *Result* and returns an XCS computation that performs the necessary adjustments to the population and time step.

```
update_this ::
  (P.Population p c a, Mutator o c a,
    RandomGen g, MonadSystem o a io) ⇒
  Result o c a → XCS io g p o c a ()
```

If the current step was an exploration step (whether that was the case can be determined via the action set's mode), the following updates are performed (the details are left out here as they do not differ from the original work [7]):

(1) update the metadata of the classifiers in the action set,
(2) perform action set subsumption (if that feature is enabled), and
(3) run the GA on the action set (if indicated).

In exploitation steps, no update is performed.

At this point, all the building blocks for a single step of a single-step XCS are defined; it consists of interacting, updating classifiers and increasing the tick counter.

```
step_single ::
  (P.Population p c a, Detector o c, Mutator o c a,
    RandomGen g, MonadSystem o a io) ⇒
  XCS io g p o c a ()
step_single = interact >>= update_this >>= increaseTime
```

A *run* of a single-step XCS simply repeats $step_{single}$ until the system signals the EOP.

```
steps_single ::
  (P.Population p c a, Detector o c, Mutator o c a,
    RandomGen g, MonadSystem o a io) ⇒
  XCS io g p o c a ()
steps_single = untilM eop step_single
```

3.9 XCS for multi-step problems

Multi-step XCS differs from single-step XCS in that the classifiers that were in the action set in the *previous* step are updated instead of the ones from the current step. In the long run, this is meant to result in a reward for a correct lookahead.

The classifiers from the previous action set are not guaranteed to exist any more one step later as several of them may have been deleted after new classifiers were inserted into the population (e. g. through covering or the GA). In particular, the set of classifiers to be updated might be empty; therefore the action set type can not be used to pass these classifiers to the next step. The *update set* thus structurally differs from the action set in the set of classifiers being possibly empty.

```
data USet c a = USet
  { mode_last      :: Mode
  , proposing_last :: [Classifier c a] }
```

A $Result_{last}$ contains all the data from the previous step that is needed to perform an update.

```
data Result_last o c a = Result_last o (USet c a) Reward
```

A multi-step XCS updates the classifiers of the *previous* update set. After that, it transforms the current action set into an update set for the next step by

- deleting the classifiers that are not in the population any more
- "refreshing" the remaining classifier's metadata to reflect the ones in the population.

```
update_last ::
  (P.Population p c a, Mutator o c a,
    RandomGen g, MonadSystem o a io) ⇒
```

$$\begin{aligned}&\mathit{Result}_{last}\ o\ c\ a \to \mathit{Result}\ o\ c\ a \to\\ &\quad \mathit{XCS}\ io\ g\ p\ o\ c\ a\ (\mathit{Result}_{last}\ o\ c\ a)\end{aligned}$$

In each step, a multi-step XCS receives the previous step's update set if it exists and returns an XCS computation that will result in its own update set (wrapped in a *Maybe* for simplifying the combination of multiple steps).

$$\begin{aligned}&\mathit{step}_{multi} ::\\ &\quad (\mathit{P.Population}\ p\ c\ a, \mathit{Detector}\ o\ c, \mathit{Mutator}\ o\ c\ a,\\ &\qquad \mathit{RandomGen}\ g, \mathit{MonadSystem}\ o\ a\ io) \Rightarrow\\ &\quad \mathit{Maybe}\ (\mathit{Result}_{last}\ o\ c\ a) \to\\ &\qquad \mathit{XCS}\ io\ g\ p\ o\ c\ a\ (\mathit{Maybe}\ (\mathit{Result}_{last}\ o\ c\ a))\end{aligned}$$

If there is no update set from the previous step (e. g. because it is the very first step), the XCS interacts with the environment and, if the end of problem is reached, performs the special end of problem update. Since no standard classifier update is involved, we can directly transform the *Result* to a Result_{last} (remember that *increaseTime* takes one parameter—here, it receives the Result_{last}—and returns it unmodified).

$$\begin{aligned}&\mathit{step}_{multi}\ \mathit{Nothing} =\\ &\quad \mathit{interact} \ggg\!= \mathit{update}_{eop} \circ \mathit{toLast} \ggg\!= \mathit{increaseTime}\end{aligned}$$

If there *is* an update set from the previous step, the XCS additionally updates its classifiers.

$$\begin{aligned}&\mathit{step}_{multi}\ (\mathit{Just}\ \mathit{last}) =\\ &\quad \mathit{interact} \ggg\!= \mathit{update}_{last}\ \mathit{last} \ggg\!= \mathit{update}_{eop} \ggg\!= \mathit{increaseTime}\end{aligned}$$

A run of a multi-step XCS consists of repeatedly applying the step_{multi} function to its own result until the system signals that the end of problem is reached.

$$\begin{aligned}&\mathit{steps}_{multi} ::\\ &\quad (\mathit{P.Population}\ p\ c\ a, \mathit{Detector}\ o\ c, \mathit{Mutator}\ o\ c\ a,\\ &\qquad \mathit{RandomGen}\ g, \mathit{MonadSystem}\ o\ a\ io) \Rightarrow\\ &\quad \mathit{XCS}\ io\ g\ p\ o\ c\ a\ (\mathit{Maybe}\ (\mathit{Result}_{last}\ o\ c\ a))\\ &\mathit{steps}_{multi} = \mathit{iterateUntilM}\ \mathit{eop}\ \mathit{step}_{multi}\ \mathit{Nothing}\end{aligned}$$

3.10 Application

Finally, we can give an—albeit abstract, for brevity's sake—example of the application of our framework. Given a system (where m is the system monad)

newtype *System x = System { runSys :: m x }*

the multi-step XCS can be applied to it like this:

$$\mathit{runSys} \circ \mathit{runXCS}\ \mathit{steps}_{multi}\ \mathit{init}\ \$\ \mathit{mkStdGen}\ 123$$

Here, *mkStdGen* is used to create a standard random generator with seed 123 whereas the *runXCS* function is a helper function for handling the monad transformer stack:

$$\mathit{runXCS}\ \mathit{xcs}\ s\ g = \mathit{runRandT}\ (\mathit{runStateT}\ (\mathit{runXCS}'\ \mathit{xcs})\ s)\ g$$

Owing to our functional approach, experiments with our framework are reproducible in an elegant way; for example,

$$\mathit{runSys} \circ \mathit{runXCS}\ \mathit{steps}_{multi}\ \mathit{init} \circ \mathit{mkStdGen} <\$> [1\,..\,100]$$

is a list of the results from 100 deterministic XCS runs with the seeds ranging from 1 to 100.

4 RELATED WORK

Before we conclude our work, we give a short overview of other important theoretic work on XCS that has not already been mentioned. The majority of these publications bases its results on experimental validation which, as stated in the introduction, will not be sufficient in the long term.

In the very first publication about XCS, Wilson investigates the *generalisation hypothesis* (pressure towards accurate but also maximally general classifiers) [23]. Kovacs postulates the *optimality hypothesis* (under certain circumstances, XCS can reliably evolve optimal populations) in his master's thesis [16].

Bull develops a Markov model for LCSs with an accurracy-based fitness measure [1]; at that, he lays more emphasis on the analysis of the GA (action selection is always done randomly).

The work by Butz et al. features a number of important insights into XCS's workings that amongst other things underpin the generalisation hypothesis. They identify different kinds of pressures that are at work in XCS [6] as well as preconditions that need to be fulfilled for XCS to learn properly [5, 6]. They also derive several bounds on the population size and show that XCS can PAC-learn k-DNF problems [3, 4].

Stalph et al. transfers a number of the theoretical findings about XCS to XCSF [21].

In order to analyse the evolutionary process in XCS, Iqbal et al. introduce parent trees through which a classifier's evolution can be traced [14]. Their results corroborate earlier work such as the generalisation hypothesis.

Nakata et al. examine how some of XCS's parameter can be set in a way such that no over-generalisation occurs [19]. In their work they also criticise that there is too little theoretic foundation to derive proper parameter settings.

5 CONCLUSION AND FUTURE WORK

In this paper we presented an alternative description of XCS: an algebraic approach through functional programming instead of the widely used algorithmic one. In the process, we implemented a general XCS framework in Haskell.

By building XCS from algebraic data types and pure functions only, the interdependencies between XCS's components can be seen more clearly. Given a problem, our type classes exactly specify the definitions necessary for applying XCS to it while the function signatures show unambiguously which part of the XCS algorithm accesses which.

The purely mathematical treatment also allowed to identify the exact components of the mutable state managed by XCS as well as a minimal set of functions it is shared between. These insights may for example be useful in the process of performance optimisations (e. g. through parallelisation).

Our future work will include an analysis of how well the different XCS derivatives can be expressed with our algebraic framework. Some will probably be straightforward to deal with (e. g. XCSF), whereas for others, further abstraction and refinement will be required.

The long-term vision is to conduct formal proofs of the properties of XCS using our algebraic description: We aim to derive problem classes and based on them analyse XCS's behaviour formally. One of the earlier steps on that path will be to revisit several of the theoretical publications about XCS. We expect to get further insights into parameter selection and computational as well as storage bounds.

ACKNOWLEDGMENTS

We would like to thank Dominik Rauh and Anthony Stein for a number of helpful discussions.

REFERENCES

[1] Larry Bull. 2002. On accuracy-based fitness. *Soft Computing* 6, 3 (2002), 154–161.

[2] Larry Bull. 2015. A brief history of learning classifier systems: from CS-1 to XCS and its variants. *Evolutionary Intelligence* 8, 2–3 (2015), 55–70.

[3] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. 2005. Computational Complexity of the XCS Classifier System. In *Foundations of Learning Classifier Systems*, Larry Bull and Tim Kovacs (Eds.). Springer, 91–125.

[4] Martin V. Butz, David E. Goldberg, Pier Luca Lanzi, and Kumara Sastry. 2007. Problem Solution Sustenance in XCS: Markov Chain Analysis of Niche Support Distributions and the Impact on Computational Complexity. *Genetic Programming and Evolvable Machines* 8, 1 (2007), 5–37.

[5] Martin V. Butz, David E. Goldberg, and Kurian Tharakunnel. 2003. Analysis and Improvement of Fitness Exploitation in XCS: Bounding Models, Tournament Selection, and Bilateral Accuracy. *Evolutionary Computation* 11, 3 (2003), 239–277.

[6] Martin V. Butz, Tim Kovacs, Pier Luca Lanzi, and Stewart W. Wilson. 2004. Toward a Theory of Generalization and Learning in XCS. *IEEE Transactions on Evolutionary Computation* 8, 1 (2004), 28–46.

[7] Martin V. Butz and Stewart W. Wilson. 2001. An Algorithmic Description of XCS. In *Advances in Learning Classifier Systems, Third International Workshop, IWLCS 2000*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer, 253–272.

[8] Haskell community. 2017. Haskell Implementations. (February 2017). Retrieved 2018-03-16 from https://wiki.haskell.org/Implementations

[9] Jan Drugowitsch. 2008. *Design and Analysis of Learning Classifier Systems - A Probabilistic Approach*. Studies in Computational Intelligence, Vol. 139. Springer.

[10] Jan Drugowitsch and Alwyn M. Barry. 2007. Mixing Independent Classifiers. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, Hod Lipson (Ed.). ACM, 1596–1603.

[11] Narayanan Unny Edakunni, Tim Kovacs, Gavin Brown, and James A. R. Marshall. 2009. Modeling UCS As a Mixture of Experts. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1187–1194.

[12] GHC Users Guide [n. d.]. Functional Dependencies. ([n. d.]). Retrieved 2018-03-16 from https://downloads.haskell.org/~ghc/8.0.2/docs/html/users_guide/glasgow_exts.html#functional-dependencies

[13] C. A. R. Hoare. 1983. An Axiomatic Basis for Computer Programming. *Commun. ACM* 26, 1 (1983), 53–56.

[14] Muhammad Iqbal, Will N. Browne, and Mengjie Zhang. 2015. Improving genetic search in XCS-based classifier systems through understanding the evolvability of classifier rules. *Soft Computing* 19, 7 (2015), 1863–1880.

[15] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.

[16] Tim Kovacs. 1996. *Evolving Optimal Populations with XCS Classifier Systems*. Master's thesis. University of Birmingham, Birmingham, UK.

[17] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Ron K. Cytron and Peter Lee (Eds.). ACM, 333–343.

[18] Simon Marlow et al. 2010. Haskell 2010 – Language Report. (2010). Retrieved 2018-03-16 from https://www.haskell.org/onlinereport/haskell2010/

[19] Masaya Nakata, Will N. Browne, Tomoki Hamagami, and Keiki Takadama. 2017. Theoretical XCS Parameter Settings of Learning Accurate Classifiers. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017*, Peter A. N. Bosman (Ed.). ACM, 473–480.

[20] Bryan O'Sullivan, John Goerzen, and Don Stewart. 2008. *Real World Haskell*. O'Reilly Media.

[21] Patrick O. Stalph, Xavier Llorà, David E. Goldberg, and Martin V. Butz. 2012. Resource management and scalability of the XCSF learning classifier system. *Theoretical Computer Science* 425 (2012), 126–141.

[22] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. ACM, 61–78.

[23] Stewart W. Wilson. 1995. Classifier Fitness Based on Accuracy. *Evolutionary Computation* 3, 2 (1995), 149–175.

[24] Brent Yorgey et al. 2018. The *MonadRandom* Haskell package. (2018). Retrieved 2018-03-22 from https://hackage.haskell.org/package/MonadRandom-0.5.1