Towards A Generic Computational Intelligence Library: Preventing Insanity

G. Pamparà, *Member, IEEE*

A.P. Engelbrecht, *Senior Member, IEEE*

***Abstract*—This paper proposes a library for computational intelligence using functional programming to address the complexities in algorithm implementation and highlighting specific concerns that are often ignored in the algorithm descriptions. Useful abstractions, common in the paradigm of functional programming, are used to make implementation specifics of algorithms part of the algorithm definition, resulting in the tracking of these effects, together with the control of the effects. Effects, requiring management within an algorithm, include the use of pseudo-random number generators, writing data to files or the console, or providing the control parameter configuration of the algorithm in an experiment. By defining the units of work for an algorithm in a general and generic form, composition of these different algorithmic units is possible, thereby creating larger, more complex computational units. The software library providing such reusable, peer-reviewed composable computational unit, is called CIlib.**

I. Introduction

The continuing growth of Computational Intelligence (CI) research results in a continual increase of different algorithms which perform well on different classes of problems. Such algorithms include artificial neural networks, evolutionary algorithms, swarm-based algorithms, fuzzy systems and artificial immune systems [1]. Moreover, research in other areas of computer science, such as programming language design, are naturally also continuing in parallel, providing improvements that are applicable to the design of CI algorithms. Not taking advantage of such improvements may result in potentially missed opportunities to express ideas in ways that may not only be new and novel, but in ways that allow for more expressiveness thereby reducing boilerplate code, which is the additional syntax and code required by the programming language in order to express the desired intention, and allowing the true intention of the program to be more obvious.

CI researchers follow a "throw away" culture whereby researchers develop an algorithm or other concept, proceed to experiment with the idea, obtain empirical evidence for the discovery, which then ultimately culminates in a conference paper or journal article. Once the research is published, and the idea is not pursued further, it often happens that all work related to producing the research output is then pushed to one side. Hopefully, the work is archived, so that upon receiving a query from a reader of the research output, the work can once again be inspected in order to clarify questions of the reader. Generally speaking, the development life cycle for a research study involves:

- design and implement an algorithm or some kind of variant,
- select appropriate benchmark problems to determine algorithm effectiveness, and
- evaluate the obtained results for the given set of benchmark problems, comparing to results obtained from a set of other algorithms.

This is a very tedious process, particularly for researchers consuming the research output. The replication of the provided empirical results may not be an obvious task. It requires a large time investment in order to correctly duplicate the results of others, allowing the comparison of the results, finally allowing the production of a research output, and then continue the cycle by archiving the research, thereby perpetuating the problem. Furthermore, in addition to the time investment required to duplicate the results, the following effects may occur:

- The published literature may neglect to correctly specify all implementation details, making the reproduction of the published results impossible because subtleties may be removed due to space constraints or are simply ignored in error.
- Errors in programming implementations remain unnoticed and may have a drastic impact on the generated results.

Preventing the additional effort to re-implement an algorithm is very beneficial. Furthermore, consolidating this effort into a common set of tools will allow for faster verification and / or the dismissal of previous results, eliminating the need for algorithm re-implementation and expensive re-computation of published results. Previously, such an effort was started by the computational intelligence research group (CIRG) at the University of Pretoria. A framework, known as the computational intelligence library (CIlib) [2], [3], [4], was created as an open-source project (`http://github.com/cirg-up/cilib`) which welcomed contributions and participation from other researchers. CIlib aimed to be a framework of components that may easily be combined and reused in order to facilitate experimentation and exploration of CI algorithms and related ideas. The framework allowed the "weaving" together of algorithm components using an XML based scripting language.

The objective of this paper is to highlight the evolution of CIlib into a piece of software that allows for faster experimentation, whilst remaining extremely principled by tracking the important effects of an algorithm in order to address specific concerns in CI research, such as the management of the pseudo-random number generator. The changes from the initial releases of CIlib have been drastic, yet simple, resulting in a better set of tools for researchers to use through the use

G. Pampará (email: gpampara@gmail.com) and A.P. Engelbrecht (email:engel@cs.up.ac.za) are both with the Department of Computer Science, University of Pretoria, Gauteng, South Africa.

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.207

of functional programming. The rest of the paper is organized as follows: Section II discusses the background of CIlib, and identifies specific problems that have necessitated change. Section III describes the benefits that Functional Programming (FP) can provide, in particular to CI algorithm development. The new design of CIlib is discussed in Section IV, contrasting it to the older framework design. The current status of the project is provided in Section V, with a discussion on planned future work given in Section VI.

II. Background

Cilib is a project that was created organically based on the requirements of CIRG, a computational intelligence research group at the University of Pretoria, consisting of a relatively large number of students that are all working on overlapping CI research areas. This overlap in research was significant enough that a common set of program code started forming a project. The project was formally started [5], [6] in order to minimize development time for all students, and to also facilitate the sharing of knowledge between research group members.

The project started using Java as the programming language of choice, due to the popularity and also due to the number of research group members who were familiar with the language. CIlib started evolving to contain implementations of algorithms that are, above everything else, generic and concise. Over time, different CI paradigms were added to the project, adding different algorithm implementations and integrating components that started to span across the different CI algorithm paradigms. As a trivial example, it became quite simple to use Particle Swarm Optimization (PSO) to train a Neural Network (NN), or to apply mutation operators to a particle swarm optimizer. Abstractions started forming that hid away very specific intricacies of algorithms allowing simpler composition of differing components.

Over time the project became more and more complex, eventually becoming a framework with an embedded language based on XML to describe experimental designs. The framework was used very successfully for many years, but users started to experience severe problems due to the fact that very specific algorithmic concerns were not addressed from project inception. Most notably, experiments started to become more difficult to reproduce as the design started hiding the pseudo-random number generator within deep object-oriented class hierarchies, preventing setting the seed value for the pseudo-random number generator in a simple way, allowing for experimental reproducability. Furthermore, deep object-oriented inheritance hierarchies posed a challenge as small changes in design started requiring considerable implementation effort due the cascading effect that a small change in a super class of the hierarchy required. Additionally, application state related errors started becoming more common, resulting in strange behavior that took considerable time and effort to both trace and resolve. The largest identified problem was actually the XML-based language that described and declared the algorithm compositions. The manner of operation of XML-based language interpreter (called the "simulator") forced a specific programming style on users for both the algorithm declaration and implementation. This style is known as the JavaBean [7] pattern, whereby every program class member required both a "getter" and a "setter" method definition. Using JavaBeans, a declared algorithm could be instantiated through the use of the reflection [8] API in Java. This decision ultimately limited the expressiveness of the framework and forced a specific flow of logic, which often resulted in the application of workarounds to achieve desired implementations. Furthermore, the interpreter for the XML-based language imposed specific behavior with respect to concurrency, often blocking to perform synchronization of runtime threads so that data may be written to disk. This synchronization effectively made program executions largely sequential, removing most of the benefit offered by parallelism.

It became clear that the current CIlib implementation was not one that could be maintained. The core development team, responsible for the framework, started to experiment with other designs and looked to other areas for inspiration. Inspiration was found, by examining current trends in programming language design and identifying that algorithms are, unsurprisingly, structures that can be reduced to be simple functions that are concerned with input and produce output. This introduced Functional Programming (FP), and as a result, allowed the usage of the many tools of FP, such as curried functions and partial application to name only two very simple elementary features.

III. Functional Programming in CI

Functional Programming (FP) [9] is a programming style that is heavily focused on the use of functions. These functions model computation as the evaluation of mathematical functions, eschewing mutable and state-changing data. Within FP, computation is declared using expressions which always return a value, whereas imperative programming is centered around the use of statements to perform computation. Because expressions return values (which are the result of the computation), expressions can easily be composed into larger expressions which declare more complex computations. As a result the expressiveness of function-based programming is rather high in contrast to an imperative style.

Examining the manner in which CI algorithms operate, it can clearly be seen that generally these algorithms require some form of input and return some output, which may or may not be different from the input data. An algorithm can therefore be seen as nothing more than a function of input to output, without concerning oneself with the internals of the algorithm. This is the fundamental underpinning of what needs to be expressed and, is the new representation for algorithm definitions within CIlib.

Function values are low-level primitives and using them directly results in program code that is fairly verbose. FP practitioners are against code repetition and have created several abstractions that remove the complexity of maintaining this verbosity. This allows the intention of the program code to be more visible and adheres to the "don't repeat yourself" or DRY principle of software engineering. It should be explicitly stated that although FP operates on the idea of simple mathematical functions from input to output, the functions themselves are pure. Purity of functions is a property whereby no additional side-effects are produced in the function evaluation and that the function operates solely on the data passed in via function arguments. External side-effects or simply effects, are exactly

what computers need to do to be useful and include things like writing strings to the console. FP programs have therefore, been extended in a variety of ways so that external effects can be tracked or maintained without breaking the purity of the function, nor function composition.

One such side-effect which is not possible in well-principled FP program code is the use of global state (or global variables). Using such state results in potentially difficult-to-understand program code, as changes in the global state may affect the running of the program in undesirable ways. This undesirable behavior is then further compounded when parallelism is taken into account, as any number of parallel processes may adjust this global state or read from it. It is, however, possible to address this concern with the use of well-established concurrency techniques, but all the techniques that enable state sharing between parallel processes are simply unnecessary when the data in question is immutable. Allowing parallel processes to read immutable data provides zero risk as the data cannot be changed by any external actor.

If the pure, lazily evaluated functional programming language Haskell [10] is taken as an example, the order of the program evaluation is not clearly defined, as differing program states may result in different evaluation paths. Haskell has a lazy evaluation model (also referred to as call-by-name) instead of being strict, which is the more common model and the model used by Java, C/C++, etc. Therefore, because the order of the program evaluation may be different based on the current program execution path, the language designers needed to introduce a structure or something similar that would allow for the sequencing of execution paths, yet at the same time, supporting the lazy evaluation model and purity that Haskell offers. Such a structure that allows for this sequencing is called the Monad, and is a computational model formulated in the category theory branch of mathematics.

From category theory, several other structure formulations have also become widely used to abstract away complexity and to categorize the functionality that the structures can provide. Such computational structures include Functor, Applicative Functor [11], and Monad [12], [13]. There are others that are also directly applicable to programming and some structures are formulated in the mathematical field of topology, such as Semigroup and Monoid [14].

These structures differ in specificity, such that the more specialized structures have fewer possible instance implementations - which is an interesting dichotomy. In order of specificity, Functor is the most general structure and Monad is the most constrained, with Applicative in between both Functor and Monad. As a result, monad has several derived operations that have proved to be very useful in general programming tasks.

Within the context of CI algorithms, there are several important aspects that need to be controlled in order to allow for a CI algorithm to execute. One such aspect is a piece of global state that is always present: the pseudo-random number generator. When a random number is sampled from such a pseudo-random number generator, the internal state of the generator needs to change so that, on the next call, a different value can be computed. This is troublesome for several reasons:

- The changing state is something that lives outside of the scope of the program.
- It is possibly difficult to ensure that this external state can be reset to a "known" value in order to reproduce a value (or set of values).
- It is unknown if this external state is operated on by other programs.
- The complexities are increased significantly if more than one pseudo-random number generator is involved.

It is therefore desirable that the pseudo-random number generator is accessible from within the program itself but be hidden from other programs. At the same time the usage of the generator should be tracked to ensure that the usage is correctly sequenced, ensuring that uncontrolled usage is not possible.

IV. Design

The original CIlib developed into a framework for running algorithms, but this restricted users in a number of ways:

- Algorithm definitions required the embedded XML language, but the XML language was not expressive enough to express all cases needed by researchers.
- Experimental result output was handled by the simulation within CIlib and that was limited to a specific format which targeted plain text files as the default.
- The framework design prevented users from using individual pieces of the framework for other uses outside of the framework itself. The framework was so tightly coupled that either the entire framework was used or none of it at all.

As a result of the above, changes were needed to address the problems associated with the usage of the software. Based on the requirements of the users and the realization that functional programming may result in better algorithmic representations and composition, the current Java implementation of CIlib was frozen and a new code base started to develop, using the programming language Scala [15]. Scala was selected for a single reason only: it compiles to the Java Virtual Machine (JVM) [16] bytecode and the JVM is a platform that all research group members were relatively comfortable with. The execution speed of the JVM is also favorable, being slower that or similar to compiled languages like C or C++, without the need to hand optimize code to achieve good performance [17]. JVM bytecode is also more portable and distributable for cluster based computation. Switching the underlying platform to be Haskell would result in a much greater impact as all development tools would need to be redefined and relearned, regardless of stronger type inference and faster execution speeds.

It was decided that the new sources would be a library first, preventing the problems identified with the framework's design, referenced in Section II. A library would allow users to decide for themselves, how to setup experiments and how to manage the produced data, by writing the data to file, persisting into a database or cloud storage. Experimental setup is now

also done directly in Scala code and the embedded language introduced to overcome limitations imposed by Java, is no longer necessary. It is certainly true that a sister project will develop that will contain common data management and setup program code as more users start experimenting with the new design. The design of the library is one that is focused on strong principles, among which are:

- **Correctness:** The correctness of an implementation is of the utmost importance and should be valued above any form of performance optimization. The intention of the implementation should always be visible so that obvious errors can be isolated and the corrected. The primary goal of CIlib is to provide a collection of program code that is simple to understand. The implementation should also be peer reviewed to ensure that the implementation is correct. Furthermore, having consensus on an implementation for a given algorithm will provide greater reuse as the same algorithm is available for use and can easily be referenced for inspection and critique later.
- **Type safety:** The usage of types within a strong type system, such as the type system provided by Scala, are advantageous as they can prevent the wrong forms of data being used. Additionally, it is important that the defined types result in incorrect program formulation to be impossible to represent. Preventing such representations then also removes the need to perform additional logical tests to ensure the validity of provided data. For example, if a function declares a positive integer as a parameter, it must be impossible to provide a negative value. Such constructions prevent error states, thereby improving the guarantees that the program code represents. This is a process of encoding invariants into the type system and types.
- **Reproducability:** Fundamentally, having an algorithm available for use is only a partial solution to a greater problem, the problem of reproducing the experimental results observed in publications. If a result set is reported, it should be acceptable that a user can then run the given algorithm, provide the expected data values such as the pseudo-random number generator seed, and obtain the exact same results as the original author. This will prevent errors in publications and furthermore provide surety that implementations are correct. To aid in the reproducability, the CIlib library project has a DOI [18] associated with the software releases, so that the exact same program code may be used when required.

The following subsections discuss individual core aspects of the functional redesign of CIlib, ensuring that the above mentioned principles are enforced.

A. Position

Candidate solutions to a given problem are described by two distinct cases within CIlib. Both these cases have the type of `Position`:

- **Point:** A single point within a multi-dimensional search space. The representation is of nothing more than a vector within the multi-dimensional space and contains no other information.
- **Solution:** A multi-dimensional search space vector together with a fitness value and a list of violated search space constraints. A Solution is more valuable than a Point as it defines a potential candidate solution to the given optimization problem, within the given search space.

It is important to distinguish between the two cases, because their meaning is fundamentally different. Furthermore, the Position cases form a closed algebra and such closed algebras are known as Algebraic Data Types (ADTs) [19].

B. Entity

Within evolutionary computation, swarm intelligence, and other paradigms, individual agents participate within the algorithm. Looking carefully at these structures, it is evident that a single common structure can be extracted. For example, an `Individual` is a structure that contains a candidate solution to the current optimization problem and an optional fitness value. Similarly, a `Particle` also contains a candidate solution, an optional fitness, a personal best solution, a velocity and a personal best fitness. There is an overlap between the particle and individual representations and similarly, this overlap is seen between agents of other algorithms. As a result, CIlib represents all these structures using a generic structure called an `Entity`, where the entity maintains a position within the search space and some state relevant to the current entity.

Constraints are placed on the algorithmic components to ensure that the provided Entity instances are of the correct form, with respect to the state that the Entity is maintaining. It wouldn't make sense to pass an Entity that does not maintain a velocity vector to an algorithm component that makes use of the velocity vector in a calculation. As such, these encoded type level constraints result in compilation failures if the wrong kind of Entity formulation is provided to library code that cannot operate on the data, ensuring correctness and consistency in the implementations. Such programming level constraints are enforced through the use of type classes [20], which are a pattern in Scala using the *implicit* language feature.

An `Entity` is defined, for a given state S and `Position` type A, as:

```
case class Entity[S,A](
  state: S, pos: Position[A])
```

C. RVar

At the core of the library is the `RVar` data type, which represents a random value or variable. `RVar` represents a random computation that has not yet been performed but has been declared, and as such means that the effect of the randomness can be tracked within the computation. `RVar` forms a monad instance and is the base monad for CIlib, but allows for various operations and has a set of probability distributions that derive from it, which include all the expected distributions used in CI research. `RVar` requires that a seeded pseudo-random number generator instance is provided upon

execution and will ensure that the generator's state is managed and threaded through the computations correctly.

`RVar` is itself a nested set of monads that manage the state of the provided pseudo-random number generator, extending the usage to be stack-safe (i.e. preventing stack overflow errors). With the ability to track and maintain randomness within an algorithm, composition of `RVar` instances allows for larger computations, with the pseudo-random number generator's state threading being addressed by the `RVar` monad. Moreover, `RVar` is a description of the randomness, without explicitly defining how randomness or which type of pseudo-random number generator is applied. Executing a `RVar` instance with the same seeded pseudo-random number generator will *always* result in the same final value and pseudo-random number generator state being the result of the computation.

D. Step

`Step` builds on `RVar` functionality and is intended to be a single operation within a CI algorithm. Examples of operations include adding noise to a vector, creating the new position for a particle by adding the old position and the new velocity vector together, or simply evaluating the fitness of the current `Entity`.

In order to achieve definitions for similar operations, two additional pieces of information are required for a `Step` definition:

- the optimization scheme (`Opt`) to use, and
- an evaluator instance (`Eval`) that can calculate the fitness value of a given position using the currently defined problem.

This results in a signature that is roughly equivalent to the following curried function:

```
Opt => Eval[A] => RVar[B]
```

where A and B, respectively, are type parameters for the type of the evaluation (e.g: $Double$) and the resultant type of the `RVar` computation. `Step` instances are then composed together to form an algorithm definition. Algorithms are currently defined to be functions that take two parameters:

- the current collection of `Entity` instances and,
- the current `Entity`

Applying these parameters to the algorithm function then results in a new `Entity` instance, wrapped within a `Step` action. The resultant `Entity` instance from the `Step` action then replaces the original `Entity` instance, which was passed to the algorithm function. The algorithm function, yielding the `Step` action, has the shape:

```
List[Entity[S,A]] =>
  Entity[S,A] =>
    Step[B,Entity[S,A]]
```

`Step` is a monad transformer [21] that stacks on top of `RVar` and, is itself, an instance of monad as well. `Step` can lift `RVar` instances into the `Step` context and as such allows for a greater amount of code reuse.

E. StepS

Many algorithms require additional data values that are used during the execution of the algorithm. Some examples of such additional data includes the ρ-value of the Guaranteed Convergence PSO (GCPSO) [22] which maintains the size of the bounding box around the current global best (gbest) particle, or a multi-objective optimization algorithm which maintains an archive of Pareto optimal solutions.

Such additions are simple to add and are represented using the `StepS` data structure, which is a layer around the standard `Step` data structure. `StepS` enriches the normal `Step` action with this additional state and is the reason for the chosen name of the data structure, which can be simplified to be "Step with state". Because `StepS` is not only a monad, but also a monad transformer like `Step`, it allows arbitrary `Step` instances to be lifted into the `StepS` context, which further allows for better code reuse.

F. Iteration Schemes

CIlib is designed around how iterations are performed, as they provide a clear separation of concerns for the algorithmic components and provide a flexible manner of execution using the per iteration approach. An alternative approach would be the use of fitness evaluations to determine when execution should be stopped, but this model provides additional complications such as if the termination occurs during the production of the next `Entity` collection, is the current collection then returned as the result, or is there some heuristic that needs to be used to "merge" current and new collections?

The manner of iteration for algorithms is something external to the algorithm definitions themselves and revolves around how the next collection of `Entity` instances is built up, using the current collection. An iteration itself can be one of two very different operating mechanisms:

- **Synchronous (sync):** the new collection of entities is built up using the current `Entity` collection and applying the algorithm function for each `Entity` instance within the collection.
- **Asynchronous (async):** the new collection of entities is built up using the partial result representing the next collection of `Entity` instances and the old collection, replacing current `Entity` instances with their new counterparts for each invocation of the algorithm for each subsequent `Entity` in the current collection, until all current `Entity` instances are replaced with new instances.

Based on the manner in which the iteration schemes operate, it should be clear that the sync strategy is one that can be (but not required to be) completely parallelized because none of the new collection entities require partial information in order to be calculated. The sync strategy is an example of how the next collection of `Entity` instances can be calculated in a map-reduce fashion where new `Entity` instances are calculated separately and then reduced into a collection. However, because the async strategy needs a partial collection result it simply cannot be made parallel as there would be a large amount of blocking on other threads which ultimately would defeat the purpose of parallelism. It is also completely

possible to mix and match iteration schemes to create a custom iteration scheme for an algorithm. An an example, suppose an iteration scheme for an algorithm is being tested that must consist of n iterations using the sync scheme, followed by m iterations of the async scheme, repeated z times. Such a formulation would be completely possible in CIlib and the construction of this custom iteration scheme simply builds on the current iteration schemes that are available.

G. Embedded language

The original XML-based scripting language resulted in several problems, but the most severe of the errors was that the XML-based scripting language was not expressive enough, resulting in many additional declarations to achieve the desired algorithm specification. In addition, because the scripting language was never compiled, nor was it type-checked, many runtime errors occurred due to typing errors from the user. Because Scala, as the underlying language is expressive already, it was prudent to reuse the language itself to define algorithms specifications.

As a result, all algorithm specifications are now also defined in Scala itself, with the added benefit that the algorithm specifications are also type-checked by the compiler and compiled. The compilation process then will result in specifications that may not be correctly formulated, from a logical point of view, but at the very least the specifications will execute. The compiler will verify that a specification is not illegal, but the experimental results produced may still be erroneous. Preventing invalid programs is not a simple task, and some invalid programs may still be represented, although the number of invalid programs that can be represented will be smaller than what would otherwise be the case.

An example usage of the language to represent an algorithm specification is provided in Listing 1, with an example of running state maintenance in Listing 2.

V. Current status

The development of CIlib continues, using FP as the medium to implement various aspects of the library. Collaboration, both internal and external, is encouraged with the project source code hosted on `github.com`. Currently, the main focus is on the core library implementations, where many are already complete, based on the current requirements from CIRG with external feedback welcomed. The project itself is divided into several sub-projects, each targeting a single concern. The current collection of sub-projects include:

- **core:** the main library containing all primary abstractions. The base code is written as generically as possible, using typeclasses to express ad-hoc polymorphism for various types that are defined for the user.
- **docs:** a documentation project containing information about the library and related sub-projects. This documentation forms the primary source of information on project internals, usage information and details on various other aspects including contribution, references, etc.
- **benchmarks:** functions used for empirical research, and comparison. Various research publications mention benchmark functions, but these functions are not accessible in a simple way. This sub-project aggregates such functions, referring to the individual publications that provided these function definitions.
- **example:** usage information and examples on how to use CIlib algorithm definitions, create new algorithms using existing components, and how to execute the definitions to obtain results. Examples are written in a literate programming style [23] in order to guide the reader through the example.
- **tests:** test program code used to verify the implementations of the code the tests target. The test code, as far as possible, is written to enforce invariant laws that the implementation code must adhere to. The tests also, as far as possible, do not verify program code using static test data and the test data is generated using a property based testing framework [24], called *ScalaCheck*.

Online services, such as continuous integration servers, have also been prepared to verify library code to be in a usable state, with no obvious errors. Errors, including logic and language level, are not always avoidable and are tracked on the Github issue page, together with the planned features and releases, available on the same site for user participation and comment.

VI. Future work

CIlib already provides support for most PSO algorithms, with support for variants in place, and the next planned, is library support for the formal inclusion of generic hyper-heuristics, multi-objective optimization and cooperative optimization algorithms. Following that, the inclusion of other core algorithms is planned including support for Genetic Algorithms (GAs), Differential Evolution (DE), other similar nature inspired evolutionary algorithms, and neural networks. Due to the initial design decision to ensure that CIlib is primarily a library first, additional sister projects are planned to cater for specific aspects that fall out of scope for the core CIlib library itself, and include the following potential sister projects:

- **Algorithmic interpretation:** includes different mechanisms to execute a common algorithm definition. This will allow for the same definition of an algorithm to be executed within static and/or dynamic environments, or potentially as a single unit within a larger algorithmic structure.
- **Visualization:** resultant data may be visualized differently based on the requirement. For example, inspection of parameter values for a given algorithm using a multi-parameter plot for large dimensional data, or visualizing particle behavior within a PSO at low dimensions.
- **Data manipulation:** analysis of resultant data is very important as it allows for comparison and principled interrogation via statistical methods. Statistical analysis methods are being used more and more in published research, so it would seem prudent to make common methods available for reuse.

```
def pso[S](
  w: Double,
  c1: Double,
  c2: Double,
  cognitive: Guide[S,Double],
  social: Guide[S,Double])(
  implicit M: Memory[S,Double],
           V: Velocity[S,Double],
           MO: Module[Position[Double],Double]
  ): List[Particle[S,Double]] => Particle[S,Double] => Step[Double,Particle[S,Double]] =
    collection => x => for {
      cog     <- cognitive(collection, x)
      soc     <- social(collection, x)
      v       <- stdVelocity(x, soc, cog, w, c1, c2)
      p       <- stdPosition(x, v)
      p2      <- evalParticle(p)
      p3      <- updateVelocity(p2, v)
      updated <- updatePBest(p3)
    } yield updated
```

Listing 1: Step based formulation of the canonical PSO algorithm

```
 def gcpso[S,F[_]:Traverse](
   w: Double,
   c1: Double,
   c2: Double,
   cognitive: Guide[S,F,Double])(
   implicit M:Memory[S,F,Double], V:Velocity[S,F,Double],mod: Module[F[Double],Double]
 ) = collection => x => {
     val S = StateT.stateTMonadState[GCParams, Step[F,Double,?]]
     val hoist = StateT.StateMonadTrans[GCParams]
     for {
       gbest   <- hoist.liftMU(Guide.gbest[S,F](collection, x))
       cog     <- hoist.liftMU(cognitive(collection, x))
       isBest  <- hoist.liftMU(Step.point[F,Double,Boolean](x.pos eq gbest))
       s       <- S.get
       v       <- hoist.liftMU(if (isBest) gcVelocity(x, gbest, w, s)
                               else stdVelocity(x, gbest, cog, w, c1, c2))
       p       <- hoist.liftMU(stdPosition(x, v))
       p2      <- hoist.liftMU(evalParticle(p))
       p3      <- hoist.liftMU(updateVelocity(p2, v))
       updated <- hoist.liftMU(updatePBest(p3))
       failure <- hoist.liftMU(Step.liftK[F,Double,Boolean](
                    Fitness.compare(x.pos, updated.pos) map (_ eq x.pos)))
       _       <- S.modify(params =>
         if (isBest) {
           params.copy(
             p = if (params.successes > params.e_s) 2.0*params.p
                 else if (params.failures > params.e_f) 0.5*params.p else params.p,
             failures = if (failure) params.failures + 1 else 0,
             successes = if (!failure) params.successes + 1 else 0)
         } else params)
     } yield updated
   }
```

Listing 2: Step with state (StepS) based formulation of the GCPSO algorithm

- **Persistence:** as previously mentioned in Section IV, resultant data persistence is the choice of the user. CIlib, itself makes no assumptions about how data is stored. The choice is the user's and as a result, various options are available depending on the need.

VII. Conclusion

This paper presents a new manner to represent CI algorithms that differs from the normal programming representation, which is the most prevalent in modern research publications, referred to as imperative programming. This alternate style of algorithm representation is based on the evaluation of mathematical functions and is known as Functional Programming. Using mathematical concepts such as monad and functor, which are directly applicable to programming, a higher level of abstraction is possible. This abstraction allows for the definition of algorithmic components that may then be composed together to build larger computations.

CIlib is a library focused on computational intelligence, allowing faster composition of individual algorithmic components to create algorithm definitions. The library is built upon solid abstractions which allow for the declaration of smaller pieces of functionality that are then composed together to declare complete algorithms. These abstractions allow for the tracking of effects throughout the execution of the algorithm, the sequencing of actions and also provide a standard nomenclature for the implementations to use. Defined algorithm compositions may then be executed using different iteration and execution strategies, depending on the goals of the user.

The development of the library continues, with several additions planned for inclusion that will complete the basic functionality. These developments will always be designed in a principled manner to ensure that the representation of the underlying ideas remains not only valid, but completely reproducible by any researcher. Reproducible results allows for better cross-validation within the larger CI community, and together with user critique and feedback, improvements will be incorporated into the project.

References

[1] A. Engelbrecht, *Computational Intelligence: An Introduction*, 2nd ed. Wiley & Sons, 2007.

[2] T. Cloete, A. Engelbrecht, and G. Pampara, "CIlib: A collaborative framework for Computational Intelligence algorithms - Part II," in *Proceedings of the IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 1764–1773.

[3] G. Pampara, A. Engelbrecht, and T. Cloete, "CIlib: A collaborative framework for Computational Intelligence algorithms - Part I," in *Proceedings of the IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 1750–1757.

[4] E. Peer, A. Engelbrecht, G. Pampara, and B. Masiye, "Ciclops: computational intelligence collaborative laboratory of pantological software," in *Proceedings of the IEEE Swarm Intelligence Symposium, 2005*, June 2005, pp. 130–137.

[5] E. Peer, "A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence," Master's thesis, University of Pretoria, 2005.

[6] E. Peer, A. Engelbrecht, G. Pampará, and B. Masiye, "CiClops: Computational Intelligence Collaborative Laboratory of Pantological Software," in *Proceedings of the IEEE Swarm Intelligence Symposium*, 2005.

[7] Oracle, "Javabeans spec," http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/.

[8] F.-N. Demers and J. Malenfant, "Reflection in logic, functional and object-oriented programming: a short comparative study," in *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 1995, pp. 29–38.

[9] J. Hughes, "Why functional programming matters," in *Research Topics in Functional Programming*, D. A. Turner, Ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990, pp. 17–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=119830.119832

[10] S. Marlow, "Haskell 2010 language report," https://www.haskell.org/definition/haskell2010.pdf.

[11] J. Gibbons and B. C. d. S. Oliveria, "The essence of the iterator pattern," *Journal of Functional Programming*, vol. 19, pp. 377–402, 2009.

[12] P. Wadler, "Comprehending monads," in *Proceedings of the ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: ACM, 1990, pp. 61–78. [Online]. Available: http://doi.acm.org/10.1145/91556.91592

[13] ——, "Monads for functional programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 24–52. [Online]. Available: http://dl.acm.org/citation.cfm?id=647698.734146

[14] B. A. Yorgey, "Monoids: Theme and variations (functional pearl)," in *Proceedings of the 2012 Haskell Symposium*, ser. Haskell '12. New York, NY, USA: ACM, 2012, pp. 105–116. [Online]. Available: http://doi.acm.org/10.1145/2364506.2364520

[15] M. Odersky and al., "An Overview of the Scala Programming Language," EPFL, Lausanne, Switzerland, Tech. Rep. IC/2004/64, 2004.

[16] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 7 Edition*, 1st ed. Addison-Wesley Professional, 2013.

[17] R. Hundt, "Loop recognition in c++/java/go/scala," in *Proceedings of Scala Days 2011*, 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[18] L. A. Davidson and K. Douglas, "Digital object identifiers: Promise and problems for scholarly publishing," *The Journal of Electronic Publishing*, vol. 4, no. 2, December 1998.

[19] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class," in *Proceedings of the Third ACM Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

[20] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 60–76. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

[21] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22Nd ACM Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 333–343. [Online]. Available: http://doi.acm.org/10.1145/199448.199528

[22] F. van den Bergh and A. Engelbrecht, "A new locally convergent particle swarm optimizer," in *Proceedings of the IEEE international conference on systems, man, and cybernetics*, vol. 7, 2002, pp. 6–9.

[23] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984. [Online]. Available: http://comjnl.oxfordjournals.org/content/27/2/97.abstract

[24] C. Amaral, M. Florido, and V. Santos Costa, "Prologcheck property-based testing in prolog," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, M. Codish and E. Sumii, Eds. Springer International Publishing, 2014, vol. 8475, pp. 1–17. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07151-0_1