Evolutionary and Swarm-Intelligence Algorithms through Monadic Composition

Gary Pamparà
Department of Industrial Engineering
Stellenbosh University, South Africa
gpampara@gmail.com

Andries P. Engelbrecht
Department of Industrial Engineering, and Computer Science Division
Stellenbosh University, South Africa
engel@sun.ac.za

ABSTRACT

Reproducible experimental work is a vital part of the scientific method. It is a concern that is often, however, overlooked in modern computational intelligence research. Scientific research within the areas of programming language theory and mathematics have made advances that are directly applicable to the research areas of evolutionary and swarm intelligence. Through the use of functional programming and the established abstractions that functional programming provides, it is possible to define the elements of evolutionary and swarm intelligence algorithms as compositional computations. These compositional blocks then compose together to allow the declarations of an algorithm, whilst considering the declaration as a "sub-program". These sub-programs may then be executed at a later time and provide the blueprints of the computation. Storing experimental results within a robust data-set file format, which is widely supported by analysis tools, provides additional flexibility and allows different analysis tools to access datasets in the same efficient manner. This paper presents an open-source software library for evolutionary and swarm-intelligence algorithms which allows the type-safe, compositional, monadic and functional declaration of algorithms while tracking and managing effects (e.g. usage of a random number generator) that directly influences the execution of an algorithm.

CCS CONCEPTS

• **Computing methodologies** → **Heuristic function construction**; **Continuous space search**; **Model verification and validation**; • **Software and its engineering** → **Software libraries and repositories**;

KEYWORDS

Functional programming, Monadic composition, Reproducible, Open-source, Evolutionary algorithm, Swarm-intelligence

ACM Reference Format:
Gary Pamparà and Andries P. Engelbrecht. 2019. Evolutionary and Swarm-Intelligence Algorithms through Monadic Composition. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326845

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326845

1 INTRODUCTION

New algorithms and techniques are prodcued as research and the application of computational intelligence (CI) continues. The modern umbrella term CI includes different algorithmic families. Algorithmic families, i.e. neural networks, evolutionary algorithms, swarm intelligence, fuzzy systems, and artifical immune systems [10]. Other research areas of computer science, especially programming language theory, are allowing for a more expressive representation of logic which is verifiable by the language compiler. This expressiveness also incorporates concepts from the field of mathematics and are directly applicable to the field of CI. Incorporating the improvements from other research areas may potentially allow for more succinct problem representations as well as tools which reduce currently identified complexity.

The current research process within CI can best be described as a "one-shot" culture with experimentation and investigation being performed solely for the purposes of publication. Once published, the work to produce the research output is often forgotten unless it is explicitly required for a subsequent publication. Researchers hope that the work which produced a publication is archived in some form, so that it may be retrieved when required. Investigating the work for a publication may provide answers to questions about unclear descriptions within a publication or questions about the process to achieve the reported results. Replication of published results is a critical part of the scientific method and serves to confirm or debunk presented results. The current research process may be outlined generally as:

- create and design a new process (algorithm or technique) that addresses a problem,
- test the proposed process by evaluating the process effectiveness on a set of benchmark problems and with competing algorithms and techniques that are accepted by the field as representative, and
- analyze the obtained results and present them to the research community through a publication.

Unfortunately, reproducing a set of results requires a large time investment and is usually challenging when considering the field of CI. The algorithms and techniques often rely on random effects in order to drive the optimization process in order to obtain a solution to an optimization problem. These random effects introduce non-determinism into operators and functions. Although the random

effects allow for algorithms to achieve performance goals, they simultaneously create unintended complexity, effectively preventing result duplication. As a result, any unintended omissions within a publication's algorithm description precludes the comparison of the original algorithm, even when non-determinism is ignored.

Removing the burden to recreate the design work of a publication in an attempt to reproduce its results is very attractive. The time investment would reduce and allow for more interesting work instead of re-implementing the work of other publications. Previously, frameworks and libraries aiming to reduce this burden for researchers have been made available for use, often as open-source software. These open-source projects cannot remove all re-implementation complexity, but can make the process far simpler. Unfortunately, the open-source software usually focuses on speeding up the process of algorithm definition, allowing for a faster result production. The improved speed of result production does not necessarily allow for result replication. Without the repeatable reproduction of results, the fundamental problem with the research process in CI remains unchanged.

2 RESULT REPRODUCIBILITY OF COMPUTATIONAL INTELLIGENCE

The reproduction of a study may be seen as more than just the reproduction of the initial data set. The reproduction should be possible for the data set as well as the publication itself. The publication manipulates the data set in order to create summary statistics represented in tables. Alternatively, the data may be represented as part of a figure or plot. Fortunately, solutions already exist that allow for the reproduction of a publication based on a given set of data. These solutions include literate programming [17] documents where the data manipulation processes are included in the document sources. When creating the final publication document the embedded logic within the publication sources are executed and the results are embedded into the publication document either as tables, figures, plots or as formatted output from the process itself. Common tools that achieve this behavior include:

- LaTeX documents with embedded `Sweave` [36] code blocks
- Markdown or LaTeX documents with embedded `R` [36] code blocks, pre-processed with a tool like `knitr` [45–47]
- `org-mode` [8, 37, 38] documents with embedded code blocks from different languages and tools

Although literate programming documents fulfill the requirement to have the publication itself reproduced, the larger and more important concern is the replication of the data set itself and the container format which stores the data set.

2.1 Data set formats

The importance of the data set itself cannot be overstated. Another equally important concern is the structure that represents the data set itself. In this case, the referred structure is the file format of the data set. The most common formats for a data set are textual. For example, representing a data set as a comma separated value (CSV) text file or have the data encoded using JavaScript Object Notation (JSON). Although such data representations are valid, the disadvantages associated with the formats negatively impact the tools that consume the data to produce derived value. The disadvantages for such formats include:

- **File size:** To represent data within textual formats, all data types are converted into strings. The textual representation results in files that can be large without an external application applying compression to the file contents. A format such as JSON only compounds this effect as all data requires a repeated key value to identify the purpose of another value.
- **Data orientation and inefficient querying:** The data items for textual formats is orientated row-wise. Each data record is written to the file as a contiguous block of text. For example, extracting the n-th column from a CSV file requires reading and / or passing over all $n - 1$ preceding columns. During the analysis of data, columns are often considered in isolation to calculate derived values such as summary statistics.
- **Absence of schema:** Textual formats may only represent a textual type of data. The implication is that during data processing, the data within the file must be interrogated in order to determine the types of data that the textual items represent. The interrogation process usually requires a parser program which attempts to guess the correct data type for a piece of data. Parsing is a slow process which requires a percentage of the data file to be processed (at a minimum) in order to assign data types to the data. Alternatively, a user may explicitly tell the parser what data types are expected. This explicit process has the drawback that the user should instead interrogate the data in some manner to determine the expected data types.

Extensions to the textual data formats have been developed to try alleviate the need to assign data types after the fact. For example, the specification for JSON-Schema [2] attempts to add a schema definition to JSON data, requiring a query of the schema to determine the expected data types. The defined schema is far smaller in size than the actual data, reducing the interrogation process time dramatically. Although a schema does remove the data type assignment problem, the data items are still stored in a row-orientated manner, resulting in wasted processing time when a single column value is required. To improve the data access speed, the data set may be stored in memory but may still prove to be too large, necessitating "on the fly" compression. Dynamic compression techniques may provide to be effective but as the size of the data set increases, the techniques fail to scale efficiently.

Columnar storage data formats achieve solutions to most of the previously mentioned disadvantages to row-orientated data formats. Columnar formats allow for efficient retrieval of a given column of data values and may also allow for compression of the columnar data. Columnar data files are akin to a table within a database:

- Data values may be accessed directly
- Columns of data types may be compressed, using a compression that is suitable to the data type
- Data can be more efficiently stored using binary schemes instead of text
- File sizes are reduced due to the use of binary and compression

- The schema for the data is implicitly available without additional processing

As data sets grow and become more common place with machine learning algorithms becoming more popular, several proposals for binary, columnar data formats have been made. Importantly, the data file format should be an open-source format to allow for better integration with analysis tools. The current recommended format is the *parquet* [1, 26] file format. Parquet files are already accepted by most data analysis systems and due to the comparably small size of the data files, the format is recommended as the container for experimental results. Providing an open-source data file format allows for simplified access to the data without necessitating specialized software tools as part of the process to generate a publication document. Furthermore, the use of open-source should not be restricted to only data file formats, but also to any tool or software used to produce data for a publication. Open-source software allows for simpler review and inspection of the logic used to produce a data set.

2.2 Computational Intelligence Software

Current tools for Computational Intelligence vary from individual software packages loaded into environments like `R` and `Matlab` to software libraries and frameworks for specific programming languages. Standalone command line interface (CLI) and graphical user interface (GUI) applications [15] for CI are also available to users and researchers. As highlighted in section 2.1, an open-source tool is far more desirable than a commercial closed-source equivalent. Open-source tools allow for complete transparency, allowing anyone to inspect or review the logic for a given algorithm or a single function.

A CI software library should adhere to the following principles, derived from the requirements for reproducible publication documents and data file formats:

- **Correctness:** Algorithm implementations should always be correct and correctness should be preferred over any performance optimization. The intent of a piece of logic should also not be opaque with clear and concise implementations being preferred. The open-source nature of the software should allow for code review at any time and should facilitate discussion in order to resolve problems. A suite of property-based tests [5, 16] to validate implementations and to prove invariants, should also be present in the software repository. Additionally, all data values should be immutable and modifications to data result in new data being created.
- **Type safety:** Usage of a strong type system is advantageous in preventing the usage of the wrong "shape" of data. An incorrect shape of data could be using an integer value when the data type restricts the valid set of values to only positive integers, or an additional piece of data in a record. A strong type system has the additional benefit of presenting such errors at the compile-time of the software and not to the execution runtime. Additionally, the usage of types within a programming language can prevent the creation of invalid data, allowing data to be validated as it is created. This prevention of invalid data creation removes invalid states from implementations resulting in more robust logic definitions and simpler testing of implementations.
- **Managed effects:** The global state available to a program by the underlying platform, upon which the program executes, could be altered by a user-defined program. Modification of this shared data should always be avoided as erroneously adjusting a shared data value may have dire consequences. Ideally, the effects of mutation (or value change) should be removed altogether but that would result in a correct program without much value. The value of a program is ultimately due to a mutation occurring as a result of the computation process. It is highly desirable to have experimental results from a simulation written to a file so that they may be analyzed later.
  Instead of preventing mutations, it would be beneficial to control and manage mutations. Effect management is not about removing the ability to do something, but rather removing the ability to do something in an unstructured way. For example, an effect requiring such management would be the random number generator in an optimization algorithm. By allowing effect management, the data flow through the logic of an algorithm is formalized, allowing for the same outputs for a given set of inputs, such as the same random number generator state.
- **Citations for software:** Software should have citation information available, allowing the reader to locate the exact software used in the generation of a data set. Specifically, a document object identifier (DOI) should be available for the software reference citation, with the DOI explicitly referring to a hash which is uniquely derived from the software source code.

3 MATHEMATICS AND FUNCTIONAL PROGRAMMING

The field of mathematics, particularly the sub-fields of topology and category theory, have direct application to programming language theory in computer science. Category theory focuses on the relations (called *morphisms*) between sets of objects (called *categories*). Importantly, category theory defines processes and procedures that are structure preserving between categories of objects.

Functional programming (FP) is a style of programming where the underlying axiomatic foundation is based on the Lambda Calculus [4]. The Lambda calculus is a current research topic within category theory. The Lambda calculus describes simple semantics for computation based on functions, where functions accept inputs and return an output. Functions may be composed (or combined) together to create new functions. Functions are also *pure*: they are only allowed to operate on the arguments provided to the function and result in a new value based on the provided arguments. Pure functions also allow for *referential transparency* in FP where a function invocation may be replaced entirely with the result of the function computation, without changing the overall behavior of a program.

Category theory [34] builds upon the axioms based on objects and morphisms to create abstractions. The abstractions from category theory translate directly into programming theory allowing

the use of abstractions that have strict laws defined. The laws are especially useful, enabling the ability to reason about programs as if they were simple equations. The reasoning process is subsequently known as *equational reasoning*. From the available abstractions within category theory, the most applicable abstractions to FP and computation [34] generally are:

- Semigroup [6, 14]
- Monoid [48]
- Functor [25]
- Applicative Functor [25]
- Monad [43, 44]

The Haskell [23] programming language introduced the monad in order to track effects within the language. Haskell is a lazily-evaluated, pure FP language and exploited the sequencing ability of monads to sequence effects in the language. The lazy-evaluated nature of Haskell allows for any order of execution as the final result is guaranteed to be the same. Monads and monadic actions define a set of laws to ensure that a monad is well-behaved, allowing for the composition of effects and enables the declaration of branching effect logic.

Monads are pure values within a programming language. Only once a monad is evaluated by the runtime system do any effects actually materialize. Because the monad is a conceptual abstraction, the application of monads is not limited solely to Haskell. The influence of moands in Haskell has resulted in the language having a succinct syntax to work with monads.

4 CILIB

From the experience gained in previous attempts to describe CI computation in software [7, 9, 30–33], a principled, type-safe, purely functional and compositional library has been developed to describe and focus on reproducible evolutionary and swarm intelligence algorithms. The sections that follow describe the individual building blocks of the library from which algorithms and algorithmic operators are built upon. The library uses the Scala programming language [28] because the developers are familiar and comfortable with the Java virtual machine (JVM) platform. At the time of project redesign, Scala was the only available language on the JVM that allowed the representation of the category theory abstractions mentioned in section 3, which requires the language compiler to understand and represent higher-kinded types (HKTs) [35]. Additionally, data is always immutable and new data value is created when modifying a value instead of mutating the data in place. It should be noted that the current version of the library is drastically different from the original Java-based version and that the scope of the library is on evolutionary and swarm-intelligence algorithms.

The sections that follow describe the formulation of required building blocks to allow for the declarative description of a population based algorithm. Population based algorithms describe algorithms that fulfill the properties common to both evolutionary and swarm intelligence algorithms.

4.1 Design

The sections that follow discuss the design of CIlib by focusing on the foundational building blocks that enable the monadic composition used by the library. Data structures representing the candidate solutions within an optimization problem search space are first discussed before describing the monadic structures from which the compositional algorithmic parts are created. Finally, the data strucutures are combined in order to produce a declarative algorithm definition which may be executed.

4.1.1 Position. The `Position` type describes a location within the optimization problem search space. `Position` is an algebraic data type (ADT) and defines a closed set of possible operations. A `Position` may be one of two possible cases:

- **Point**, which is an optimization problem candidate solution which is located within the problem space, but has not yet been quantified. The candidate solution is a multi-dimensional vector which is representative of the optimization problem.
- **Solution:** When the quality of a `Position` is evaluated, the resulting quantification value is combined with the current `Point` to create a `Solution`.

When evaluating a `Position` that already has a value for the solution quality, the evaluation process does not re-evaluate the `Position`. The algebra of operations defined for `Position` vectors allows for the addition, subtraction and scaling of the `Position` values. These operations create new `Position` values using the `Point` data constructor as any movement of the candidate solution invalidates the solution quality because the candidate solution represents a different location within the optimization problem search space.

4.1.2 Entity. Evolutionary and swarm intelligence algorithms locate new candidate solutions as they execute. Literature for algorithms assigns a name, based on some metaphor, to each of these algorithm participants which the algorithm is maintaining within its population. For example, *particles* are used in a particle swarm optimization (PSO) swarm and *individuals* in a genetic algorithm (GA) population. Ignoring the metaphor for each algorithm, the algorithm participants may collectively be referred to as *entities* and groups of entities are simply referred to as a *collection*.

An `Entity` builds upon the `Position` by associating algorithmic specific memory and behaviors for an entity. An example of an entity is the entity used within a PSO which maintains additional vectors for the current *velocity* and the previous *best position*. Allowing for a general structure that can maintain any amount of extra state information creates the `Entity`, represented as:

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

where `S` and `A` are type parameters allowing the `Entity` to maintain *any* type of value for the state type `S` and *any* type for dimensions of the optimization problem.

4.1.3 RVar. The most important effect to manage within an optimization algorithm is the use of the random number generator. Most platforms provide a default global random number generator but it can be argued that this practice is frought with errors with default generators often failing randomness tests such as *TestU01* [18] (also known as the "Crush" tests) and the *DieHard* [24] tests. The

test suites determine if a random number generator is statistically random for the number of state bits the generator maintains. The system wide platform random number generator may not provide sufficient quality for scientific work.

As the tracking of the randomness effect is crucial to allow for reproducible experimental work, an abstraction to represent values with randomness applied was required. The resulting abstraction is a data structure known as `RVar` and is implemented as a state monad [43, 44], specialized to manage the state of the random number generator.

The monadic structure of `RVar` allows for a declarative description of applying randomness to a value without actually performing the action immediately. The state monad ensures that the state of the random number generator is correctly threaded through all composed computations. `RVar` values are pure until they are executed by providing a seeded random number generator to the `RVar` instance, upon which the effect is observed. Executing a `RVar` results in a single value that has randomness applied, together with the modified subsequent state of the random number generator.

4.1.4 Step. Most operators within an optimization algorithm make use of the following components:

- The source of random numbers
- A function that quantifies candidate solutions
- The strategy for the optimization (minimization or maximization) of the objective function

Although operators need not make use of all algorithm components, the components should be available. Building on the `RVar` monad allows for the definition of the `Step` abstraction. Algorithms are conceptually a sequence of steps and the `Step` abstraction provides the structure to allow for the composition of algorithm logic to build up a complete algorithm definition.

Evaluation of the quality of a candidate solution depends on the defined objective function of the optimization problem. An interface known as `Eval` provides a generalized view to the optimization problem's objective function. The optimization strategy is another algorithm configuration value, represented by the `Opt` algebraic type. The `Step` abstraction is a function which accepts an `Eval` and an `Opt` value, as the environment for the algorithm execution, and produces an `RVar`:

```
final case class Environment[A](
  opt: Opt,
  eval: Eval[NonEmptyList,A]
)

final case class Step[F[_],A,B](
  run: Environment[A] => RVar[B]
)
```

where `A` and `B` define type parameters which specify the type of the optimization problem dimensions and the result of the evaluation process. `F[_]` is a kind that takes another type as a parameter to create a type. For example, programming languages with "generics" allow specifying what the elements in a list of elements may be, i.e. a list of doubles. Such types are known as higher-kinded types (HKTs) [35] and the syntax generalizes over all HKTs that accept a single type parameter. Examples of parameteric types which "fit" into `F[_]` include:

- `List`
- `Set`
- `RVar`

`Step` has an instance of monad and is also a monad transformer [19] which stacks on top of `RVar`. Instances of `RVar` may be "lifted" into the `Step` transformer stack.

4.1.5 StepS. As part of algorithm execution, some algorithms necessitate the maintenance of additional algorithm specific state during execution. Examples of such algorithms are the guaranteed convergence PSO (GCPSO) [39] which maintains a dynamic bounding box around the best entity in the collection and multi-objective optimization algorithms where an archive of non-dominated solutions (the Pareto optimal set) are maintained as the final solution to the multi-objective optimization problem.

`StepS` is a monad transformer (much like `Step`) that allows for the persistence of an addition state value across the monad transformer stack. This algorithmic state is defined using a type parameter on `StepS`, allowing any user-defined value to be used as the state for the algorithm.

4.1.6 Iteration schemes. Evolutionary and swarm-intelligence algorithms are defined as processes that replace a target entity in the collection with a new entity. The replacement is new data and is immutable. As a result, an algorithm is regarded as a function from the current entity collection and target entity, to a new entity within a `Step`:

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

The algorithm formulation above specifies the process to produce a single new entity. Thanks to `Step` having an instance of monad, several monadic combinators are available which allow for the application of the algorithm function on the entire entity collection. Consequently, the algorithm formulation allows for isolated entity collections which exist in parallel before the new collection replaces the current collection in subsequent algorithm iterations. Measurements between the entity collections is much simpler than in the situation where the entity collection was altered in place.

Algorithm iteration may be:

- **Synchronous:** Traversing the current entity collection and applying the algorithm to each individual entity builds up the new entity collection. The entity collection remains unchanged in the algorithm usage with the new entity collection used in the following iterations. The synchronous iteration scheme is also known as the "generational" replacement scheme.
- **Asynchronous:** The new entity collection is built up using the union of the already replaced entities and the remaining entities of the current entity collection. As the traversal of the entities proceeds, the entity collection which the algorithm may reference changes as well. The asynchronous iteration may also be referred to as the "steady-state" replacement scheme.

Based on the description of how the algorithm execution occurs, it should be noted that the asynchronous iteration is dependent on the currently built up entity collection, whereas the synchronous version does not have this dependence. Opportunities to add parallelism to the traversal process within an iteration is *only* possible for the synchronous scheme because the same value for the entity collection is used.

The iteration scheme for an algorithm is not restricted to a single choice either. It is possible to create a heterogeneous iteration scheme where the basic iteration schemes are mixed. An example might be to run a synchronous iteration scheme for n iterations, followed by an asynchronous iteration for m iterations, all repeated z times.

4.2 Current Status

CIlib [29] currently supports the following algorithmic families:

- GA
- Differential evolution (DE)
- PSO
- Estimation of distribution (EDA)
- Multi-objective optimization (MOO)
- Hyper-heuristics
- Constraint handling
- Static and Dynamic optimization problems and algorithms

In addition to the necessary work to declare the above mentioned algorithms, a data output library (which uses parquet files), and an algorithm execution framework (intended to only demonstrate how to execute algorithms) are also available. Current users, however, do report that the execution framework is good enough for the majority of the use-cases that exist.

A sister library, *benchmarks* [12], also exists, providing a large collection of the most popular benchmark functions to test algorithms on. The benchmarks project relies on the core data structures (such as `RVar`) for the benchmark implementations. The available benchmark functions include both dynamic and static optimization problems and constrained problems.

Another project, *FLA* [13], is also derived from the core data structures of the library and allows the measurement of *fitness landscape analysis* [20–22, 40–42] metrics on the search space of an optimization problem.

4.3 Example Usage

To demonstrate the benefits of algorithmic declaration using monadic composition, listings 2 and 3 show the Scala code required to declare the GCPSO and an algorithm pipeline which alternates between PSO and DE to iterate an entity collection. The program code samples have had all module imports excluded for brevity, but are available as examples in the project repository.

As mentioned in section 4.1.5, the GCPSO algorithm maintains state during the algorithm execution and requires the usage of the `StepS` monad transformer. Listing 1 describes the full experimental definition for the G24 [27] set of dynamic constrained optimization problems. Listing 2 defines the GCPSO implementation with monadic actions defined in `Step` lifted into `StepS` through the `StepS.pointS` function.

Listing 3 demonstrates the composition of PSO and DE into a single algorithm. The pre-condition is that the `Entity` should fulfill all the constraints for the functions to allow the program to compile. Exploiting the Scala language feature whereby the compiler can provide *implicit* evidence, the library uses the *type-class pattern* and *lenses* [11] to prevent invalid usage by forcing a compilation error when a usage constraint is not met. The state maintained by the `Entity` must provide the ability to extract a previous `Position` and velocity in order to be used within functions designed for PSO usage, whereas DE functions ignore the `Entity` state. This behavior is useful with hyper-heuristic [3] algorithms where different heuristics are combined or swapped in and out as algorithm execution proceeds.

5 CONCLUSION

This paper described an open-source software library allowing for the compositional, type-safe and purely functional declaration of algorithms. The library focuses on enabling reproducible results by modeling algorithms as units of computations that behave like pure functions where the same output will always result from the same inputs. The library addresses the concerns for computational intelligence (CI) software by ensuring a transparent and open code-base for anyone to inspect and/or critique. The library implementation uses techniques afforded by a strongly typed functional programming language to manage and track effects which alter the behavior of a CI algorithm during its execution (such as the source of randomness) and to validate and prove implementation correctness.

Data from the algorithm execution may be captured using an efficient open-source data file format that easily integrates with existing analysis tools. These analysis tools then integrate into workflows to construct research outputs for publication. Reproduction of results is further simplified by versioning the software library with DOI references which ensures that the same base software is always available. By focusing on the reproduction of results, it is hoped that this software library will start to change how experimental work is presented and reported within publications.

Future work includes the expansion of the library and suite of benchmarks. Planned additions to the open-source software library include the incorporation of additional algorithmic families such as more complex multi-objective optimization, niching algorithms, evolutionary strategies (ES), and genetic programming (GP) and decomposition-based algorithms.

REFERENCES

[1] 2013. Apache Parquet. https://parquet.apache.org/. Accessed: 2019-01-21.
[2] 2018. JSON Schema. https://json-schema.org/. Accessed: 2019-02-11.
[3] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (01 Dec 2013), 1695–1724. https://doi.org/10.1057/jors.2013.71
[4] Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337
[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266
[6] A.H. Clifford and G.B. Preston. 1961. *The Algebraic Theory of Semigroups.* Number pt. 1 in Mathematical Surveys and Monographs. American Mathematical Society. https://books.google.co.za/books?id=4vXG2rjCUmUC
[7] T. Cloete, A.P. Engelbrecht, and G. Pampara. 2008. CIlib: A collaborative framework for Computational Intelligence algorithms - Part II. In *Proceedings of the IEEE*

```
object ExperimentsG24 {
  val envChange = Env.frequency(100)

  val problems =
    List(
      G24DCOPs.instance01(envChange, 0.25, 20.0),
      G24DCOPs.instance02(envChange, 0.25, 20.0),
      G24DCOPs.instance03(envChange, 0.25, 20.0),
      G24DCOPs.instance04(envChange, 0.25, 20.0),
      G24DCOPs.instance05(envChange, 0.25, 20.0),
      G24DCOPs.instance06(envChange, 0.25, 20.0),
      G24DCOPs.instance07(envChange, 0.25, 20.0),
      G24DCOPs.instance08(envChange, 0.25, 20.0),
      G24DCOPs.instance09(envChange, 0.25, 20.0),
      G24DCOPs.instance10(envChange, 0.25, 20.0),
      G24DCOPs.instance11(envChange, 0.25, 20.0),
      G24DCOPs.instance12(envChange, 1.0, 20.0),
      G24DCOPs.instance13(envChange, 1.0, 20.0),
      G24DCOPs.instance14(envChange, 1.0, 20.0),
      G24DCOPs.instance15(envChange, 1.0, 20.0),
      G24DCOPs.instance16(envChange, 0.25, 20.0),
      G24DCOPs.instance17(envChange, 0.25, 20.0),
      G24DCOPs.instance18(envChange, 0.25, 20.0)
    )

  val collection =
    Position.createCollection(x => Entity((), x))(
      G24DCOPs.domain, 40)

  val algs = List(
    Runner.staticAlgorithm("RIGA",
      RIGA.riga(
        bounds = G24DCOPs.domain,
        p_m = 0.3,
        p_c = 0.1,
        p_im = 0.3
      )))

  val comparison = Comparison.quality(Min)
  val rngs: List[RNG] = RNG.initN(50, 123456789L)
  def constFst(
    x: NonEmptyList[Entity[Unit,Double]],
    y: cilib.Eval[scalaz.NonEmptyList,Double]) = RVar.pure(x)

  final case class Measurements(
    globalMax: Option[Double],
    bestFitness: Double,
    rawError: Option[Double],
    cumulativeMeanError: Option[Double],
    bestErrorBeforeChange: Option[Double],
    bestErrorAfterChange: Option[Double],
    meanDiversity: Double,
    medianDiversity: Double)

  def main(args: Array[String]): Unit = {
    val process =
      for {
        a <- Process.emitAll(algs)
        p <- Process.emitAll(problems)
        r <- Process.emitAll(rngs)
      } yield
        Runner.foldStep(comparison, r, collection, a, p, constFst)
          .pipe(Runner.measure(measurementFunc))
          .take(1000)

    merge.mergeN(4)(process) // Parallelism
      .to(parquetSink(new java.io.File("riga-test.parquet")))
      .run.unsafePerformSync
  }
```

Listing 1: Dynamic constrained function experiment definition

International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). 1764–1773. https://doi.org/10.1109/IJCNN.2008.4634037

[8] Carsten Dominik. 2010. *The Org-Mode 7 Reference Manual: Organize Your Life with GNU Emacs*. Network Theory, UK. with contributions by David O'Toole, Bastien Guerry, Philip Rooke, Dan Davison, Eric Schulte, and Thomas Dye.

[9] Juan J. Durillo and Antonio J. Nebro. 2011. jMetal: A Java Framework for Multi-objective Optimization. *Adv. Eng. Softw.* 42, 10 (Oct. 2011), 760–771. https://doi.org/10.1016/j.advengsoft.2011.05.014

[10] A.P. Engelbrecht. 2007. *Computational Intelligence: An Introduction* (second ed.). Wiley & Sons.

[11] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. *SIGPLAN Not.* 40, 1 (Jan. 2005), 233–246. https://doi.org/10.1145/1047659.1040325

[12] R. Garden and G. Pamparà. 2017. A collection of n-dimensional functions. Retrieved March 2, 2019 from https://github.com/cirg-up/benchmarks

[13] R. Garden and G. Pamparà. 2017. A collection of traversal algorithms and function metrics used in Fitness Landscape Analysis. Retrieved March 2, 2019 from https://github.com/cirg-up/fla

[14] P.A. Grillet. 1995. *Semigroups: An Introduction to the Structure Theory*. Taylor & Francis. https://books.google.co.za/books?id=yM544W1N2UUC

[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[16] John Hughes. 2010. Software Testing with QuickCheck. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School (CEFP'09)*. Springer-Verlag, Berlin, Heidelberg, 183–223. http://dl.acm.org/citation.cfm?id=1939128.1939134

[17] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97 arXiv:http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html

[18] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Softw.* 33, 4, Article 22 (Aug. 2007), 40 pages. https://doi.org/10.1145/1268776.1268777

[19] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[20] K.M. Malan and A.P. Engelbrecht. 2014. Characterising the Searchability of Continuous Optimisation Problems for PSO. *Swarm Intelligence* 8, 4 (01 Dec 2014), 275–302. https://doi.org/10.1007/s11721-014-0099-x

[21] K.M. Malan and I. Moser. 2018. Constraint Handling Guided by Landscape Analysis in Combinatorial and Continuous Search Spaces. (03 2018), 1–23.

[22] K. M. Malan, J. F. Oberholzer, and A. P. Engelbrecht. 2015. Characterising constrained continuous optimisation problems. In *Proceedings of the IEEE Congress on Evolutionary Computation*. 1351–1358. https://doi.org/10.1109/CEC.2015.7257045

[23] S. Marlow. [n. d.]. Haskell 2010 Language Report. https://www.haskell.org/definition/haskell2010.pdf.

[24] G. Marsaglia. 1985. The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness. produced at Florida State University under a grant from The National Science Foundation, 1985. Access available at http://www.stat.fsu.edu/pub/diehard.

[25] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18 (01 2008), 1–13. https://doi.org/10.1017/S0956796807006326

[26] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*. 330–339. http://www.vldb2010.org/accept.htm

[27] T.T. Nguyen and X. Yao. 2012. Continuous Dynamic Constrained Optimization - The Challenges. *IEEE Transactions on Evolutionary Computation* 16, 6 (Dec 2012), 769–786. https://doi.org/10.1109/TEVC.2011.2180533

[28] Martin Odersky and al. 2004. *An Overview of the Scala Programming Language*. Technical Report IC/2004/64. EPFL, Lausanne, Switzerland.

[29] Gary Pamparà. 2007. CIlib: Typesafe, purely functional Computational Intelligence. Retrieved March 2, 2019 from https://github.com/cirg-up/cilib

[30] G. Pampara, A.P. Engelbrecht, and T. Cloete. 2008. 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)*. 1750–1757. https://doi.org/10.1109/IJCNN.2008.4634035

[31] G. Pamparà, F. Nepomuceno, and B. Leonard. 2014. CIlib. https://doi.org/10.5281/zenodo.12371

[32] E.S. Peer. 2005. *A Serendipitous Software Framework for Facilitating Collaboration in Computational Intelligence*. Master's thesis. University of Pretoria.

[33] E.S. Peer, A.P. Engelbrecht, G. Pampara, and B.S. Masiye. 2005. CiClops: computational intelligence collaborative laboratory of pantological software. In *Proceedings of the IEEE Swarm Intelligence Symposium, 2005.* 130–137. https://doi.org/10.1109/SIS.2005.1501612

[34] B.C. Pierce, B. C, B.C. Pierce, M.R. Garey, and A. Meyer. 1991. *Basic Category Theory for Computer Scientists*. MIT Press. https://books.google.co.za/books?

```
  def gcpso[S](w: Double, c1: Double, c2: Double, cognitive: Guide[S, Double])(
     implicit M: HasMemory[S, Double],
              V: HasVelocity[S, Double],
              S: MonadState[StepS[Double, GCParams, ?], GCParams])
  : NonEmptyList[Particle[S, Double]] => Particle[S, Double] =>
      StepS[Double, GCParams, Particle[S, Double]] =
   collection =>
     x => {
       val g = Guide.gbest[S]
       for {
         gbest <- StepS.pointS(g(collection, x))
         cog <- StepS.pointS(cognitive(collection, x))
         isBest <- StepS.pointS(Step.pure[Double, Boolean](x.pos eq gbest))
         s <- S.get
         v <- StepS.pointS(
           if (isBest) gcVelocity(x, gbest, w, s)
           else stdVelocity(x, gbest, cog, w, c1, c2))
         p <- StepS.pointS(stdPosition(x, v))
         p2 <- StepS.pointS(evalParticle(p))
         p3 <- StepS.pointS(updateVelocity(p2, v))
         updated <- StepS.pointS(updatePBest(p3))
         failure <- StepS.pointS(
           Step.withCompare[Double, Boolean](
             Comparison.compare(x.pos, updated.pos).andThen(_ 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: GCPSO algorithm definition

```
object Mixed extends SafeApp {
  val bounds = Interval(-5.12, 5.12) ^ 30
  val env = Environment(
    cmp = Comparison.dominance(Min),
    eval = Eval.unconstrained((x: NonEmptyList[Double]) => Feasible(spherical(x))))

  // Define the DE
  val de = DE.de(0.5, 0.5, DE.randSelection[Mem[Double], Double], 1, DE.bin[Position, Double])

  // Define a standard gBest-PSO
  val cognitive = Guide.pbest[Mem[Double], Double]
  val social = Guide.gbest[Mem[Double]]
  val gbestPSO = pso.Defaults.gbest(0.729844, 1.496180, 1.496180, cognitive, social)

  // The entity collection is the maximal set of features needed for the state.
  // In the case of DE and PSO, the particle state requires management
  val swarm =
    Position.createCollection(
      PSO.createParticle(x => Entity(Mem(x, x.zeroed), x)))(bounds, 20)

  val combinedAlg: NonEmptyList[Entity[Mem[Double], Double]] =>
                     Entity[Mem[Double], Double] =>
                       Step[Double, Entity[Mem[Double], Double]] =
    collection =>
      x =>
        for {
          a <- gbestPSO(collection)(x)
          b <- de(collection)(a)
          // Position change: current pbest is no longer valid
          c <- pso.PSO.updatePBest(b)
        } yield c

  val alg = Iteration.sync(combinedAlg)

  override val runc: IO[Unit] =
    putStrLn(Runner.repeat(1000, alg, swarm).run(env).run(RNG.fromTime).toString)
}
```

Listing 3: Pipeline of gbestPSO and DE for hyper-heuristics

id=ezdeaHfpYPwC
[35] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
[36] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/
[37] E. Schulte and D. Davison. 2011. Active Documents with Org-Mode. *Computing in Science Engineering* 13, 3 (may-june 2011), 66 –73. https://doi.org/10.1109/MCSE.2011.41
[38] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. 2012. A Multi-Language Computing Environment for Literate Programming and Reproducible Research. *Journal of Statistical Software* 46, 3 (25 1 2012), 1–24. http://www.jstatsoft.org/v46/i03
[39] Frans van den Bergh and AP Engelbrecht. 2002. A new locally convergent particle swarm optimizer. In *Proceedings of the IEEE international conference on systems, man, and cybernetics*, Vol. 7. 6–9.
[40] V.K. Vassilev. 2000. *Fitness Landscapes and Search in the Evolutionary Design of Digital Circuits*. Ph.D. Dissertation. Napier University.
[41] V. K. Vassilev, T. C. Fogarty, and J. F. Miller. 2000. Information Characteristics and the Structure of Landscapes. *Evol. Comput.* 8, 1 (March 2000), 31–60. https://doi.org/10.1162/106365600568095
[42] V. K. Vassilev, T. C. Fogarty, and J. F. Miller. 2003. *Smoothness, Ruggedness and Neutrality of Fitness Landscapes: from Theory to Application*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–44. https://doi.org/10.1007/978-3-642-18965-4_1
[43] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592
[44] Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, UK, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146
[45] Yihui Xie. 2014. knitr: A Comprehensive Tool for Reproducible Research in R. In *Implementing Reproducible Computational Research*, Victoria Stodden, Friedrich Leisch, and Roger D. Peng (Eds.). Chapman and Hall/CRC. http://www.crcpress.com/product/isbn/9781466561595 ISBN 978-1466561595.
[46] Yihui Xie. 2015. *Dynamic Documents with R and knitr* (2nd ed.). Chapman and Hall/CRC, Boca Raton, Florida. https://yihui.name/knitr/ ISBN 978-1498716963.
[47] Yihui Xie. 2018. *knitr: A General-Purpose Package for Dynamic Report Generation in R*. https://yihui.name/knitr/ R package version 1.20.
[48] Brent A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/2364506.2364520