The Journal of Open Source Software

mathlib: A Scala package for readable, verifiable and sustainable simulations of formal theory

Mark Blokpoel [1]

1 Donders Institute for Brain, Cognition, and Behaviour, Radboud University, The Netherlands

DOI: 10.21105/joss.06049

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @larkz
- @stephenfmann
- @drussellmrichie

Submitted: 24 August 2023
Published: 25 July 2024

Summary

Formal theory and computational modeling are critical in cognitive science and psychology. Formal systems (e.g., set theory, functions, first-order logic, graph theory) allow scientists to 'conceptually analyze, specify, and formalize intuitions that otherwise remain unexamined' (Guest & Martin, 2021). They make otherwise underspecified theories precise and open for critical reflection (van Rooij & Baggio, 2021). A theory can be formally specified in a computational model using mathematical concepts such as set theory, graph theory, and probability theory. The specification is often followed by analysis to understand precisely what assumptions and consequences the formal theory entails. An important method of analysis is computer simulation, which allows scientists to explore complex model behaviours and derive predictions that otherwise cannot be analytically derived[1].

`mathlib` is a library for Scala (Odersky, 2008) supporting functional programming that resembles mathematical expressions such as set theory and graph theory. This library was developed to complement the theory development method outlined in the open education book Theoretical modeling for cognitive science and psychology by Blokpoel & van Rooij (2021). To date `mathlib` is the only library that facilitates implementing computational-level simulations for fully specified formal theories.

The goal of this library is to help users to implement simulations of their formal theories. Code written in Scala using `mathlib` is:

- easy to **read**, because `mathlib` syntax closely resembles mathematical notation
- easy to **verify**, by proving that the code exactly implements the theoretical model (or not)
- easy to **sustain**, as older versions of Scala and `mathlib` can easily be run on newer machines

Statement of need

`mathlib` supports scholars in writing **readable** and **verifiable** code. Writing code is not easy, writing code for which we can know that it computes what the specification (i.e., the formal theory) states is even harder. Given the critical role of theory and computer simulations in cognitive science, it is important that scholars can verify that the code does what the authors intend it to do. This can be facilitated by having a programming language where the syntax and semantics closely matches that of the specification. Since formal theories are specified using mathematical notation (Blokpoel & van Rooij, 2021; Guest & Martin, 2021; Marr, 1982), functional programming languages bring a lot to the table in terms of syntactic and semantic resemblance to mathematical concepts and notation. `mathlib` adds mathematical concepts

[1]Computer simulations can also help scientists discover properties of the model that they would not have thought to analytically derive, even when in principle the property can be analytically derived.

and notation to the functional programming language Scala (Odersky, 2008). At the time of writing, the current version (0.9.1) supports set theory and graph theory. To appreciate the readability of functional code, relative to the formal specification, consider the following two code snippets. Both implement the same mathematical expression $\arg\max_{a \in A} f(a)$, where A is a set of strings and $f(.)$ counts the length of each string. In both snippets the set A is translated to `words` to comply with default Scala style. A non-functional implementation could look like this, where semantics (i.e., the meaning or function of the code) is more difficult to analyze due to its use of mutable variables and a loop.

```scala
def f(a: String): Int = a.length

def expression(words: Set[String]): String = {
  var maxLength: Int = 0
  var longestWord: String = ""
  for(word <- words) {
    if(f(word) > maxLength) {
      maxLength = f(word)
      longestWord = word
    }
  }
  longestWord
}

expression(Set("a", "aa"))
```

A functional implementation leveraging `mathlib` can look like this, which closely resembles the mathematical expression in form and function.

```scala
def f(a: String): Int = a.length

def expression(words: Set[String]): String = {
  argMax(words, f _)
    .random.get
}

expression(Set("a", "aa"))
```

In the next section, we provide two concrete examples to illustrate how Scala and `mathlib` make code more accessible to verify the relationship between simulation and theory.

`mathlib` and Scala support scholars in writing **sustainable** code. It is important that academic contributions remain accessible for reflection and critique. This includes simulations that also have theoretical import. Simulation results may need to be verified or future scholars may wish to expand upon the work. It is not sufficient to archive code, because in practice programming contributions in academia are easily lost because of incompatibility issues between older software and newer operating systems. Scala (and consequently `mathlib`) runs on the Java Virtual Machine (JVM) and has state-of-the-art versioning. The programmer can specify exactly which version of Scala and `mathlib` needs to be retrieved to run the code on any system that supports the JVM. Even when newer versions of Scala or `mathlib` may potentially break older code, this versioning system allows future users to easily run, adapt or expand older code.

`mathlib` is unique because its design encourages the computational cognitive scientist to write readable, verifiable, and sustainable code for simulations of formal theories. This is not to say that one cannot apply these principles in other languages, but it may require building the mathematical infrastructure that `mathlib` supports. `mathlib` differs from other libraries in that it focuses on usability and transparency for simulations of formal theories specifically, whereas other libraries that implement similar functionality focus on computational expressiveness and efficiency.

Illustrations

We present two illustrations to show the relationship between simulations implemented in Scala and mathlib and formal theories.

Illustration 1: Subset choice

The following formal theory is taken from the textbook by Blokpoel & van Rooij (2021). It specifies people's capacity to select a subset of items, given the value of individual items and pairs. For more details on this topic, see Chapter 4 of the textbook.

Subset choice

Input: A set of items I, a value function for single items $v : I \to \mathbb{Z}$ and a binary value function for pairs of items $b : I \times I \to \mathbb{Z}$.

Output: A subset of items $I' \subseteq I$ (or $I' \in \mathcal{P}(I)$) that maximizes the combined value of the selected items accordingly, i.e., $\arg\max_{I' \in \mathcal{P}(I)} \sum_{i \in I'} v(i) + \sum_{i,j \in I'} b(i,j)$.

Assuming familiarity with the formal theory, the mathlib implementation and Table 1 below illustrate how to interpret and verify the code relative to the mathematical expressions in the formal theory. In this code illustration, the following functionality is provided by mathlib: sum(., .), argMax(., .) and powerset(.). A demo of this code can be found in mathlib.demos.SubsetChoice.

```
type Item = String

def subsetChoice(
                  items: Set[Item],
                  v: Item => Double,
                  b: (Item, Item) => Double
                ): Set[Item] = {

  def value(subset: Set[Item]): Double =
    sum(subset, v) + sum(subset.uniquePairs, b)

  val allOptimalSolutions = argMax(powerset(items), value)
  allOptimalSolutions.random.get
}
```

Table 1: Mappings between formal expression and mathlib implementation.

Formal expression	mathlib implementation and description
n.a.	`Item` *Custom type for items.*
I	`items: Set[Item]` *A set of items.*
$v : I \to \mathbb{Z}$	`v: (Item => Double)` *Value function for single items.*
$b : I \times I \to \mathbb{Z}$	`b: ((Item, Item) => Double)` *Value function for pairs of items.*
n.a.	`def value(subset: Set[Item]): Double` *Function wrapper for the combined value of a subset.*
$\sum_{i \in I'} v(i)$	`sum(subset, v)`

Formal expression	`mathlib` implementation and description
	Sum of single item values, where subset is I'.
$\sum_{i,j\in I'} b(i,j)$	`sum(subset.uniquePairs, b)` *Sum of pair-wise item values, where `uniquePairs` generates all pairs `(x, y)` in `subset` with `x!=y`.*
$\arg\max_{I'\in\mathcal{P}(I)} \cdots$	`argMax(powerset(items), value)` Returns the element from the powerset of items that maximizes `value`.

Illustration 2: Coherence

Coherence theory (Thagard & Verbeurgt, 1998) aims to explain people's capacity to infer a consistent set of beliefs given constraints between them. For example, the belief 'it rains' may have a negative constraint with 'wearing shorts'. To believe that it rains and not wearing shorts is consistent, but to believe that it rains and to wear shorts is inconsistent. In case of consistency, the constraint is said to be *satisfied*. Coherence theory conjectures that people infer truth-values for their beliefs so as to maximize the sum of weights of all satisfied constraints. For a more detailed introduction to Coherence theory, see (Thagard & Verbeurgt, 1998) and Chapter 5 in (Blokpoel & van Rooij, 2021)

COHERENCE

Input: A graph $G = (V, E)$ with vertex set V and edge set $E \subseteq V \times V$ that partitions into positive constraints C^+ and negative constraints C^- (i.e., $C^+ \cup C^- = E$ and $C^+ \cap C^- = \emptyset$) and a weight function $w : E \to \mathbb{R}$.

Output: A truth value assignment $T : V \to \{true, false\}$ such that $Coh(T) = Coh^+(T) + Coh^-(T)$ is maximum. Here,

$$Coh^+(T) = \sum_{(u,v)\in C^+} \begin{cases} w((u,v)) \text{ if } T(u) = T(v) \\ 0 \text{ otherwise} \end{cases}$$

and

$$Coh^-(T) = \sum_{(u,v)\in C^-} \begin{cases} w((u,v)) \text{ if } T(u) \neq T(v) \\ 0 \text{ otherwise} \end{cases}$$

Assuming familiarity with the formal theory, the `mathlib` implementation and Table 2 below illustrate how to interpret and verify the code relative to the mathematical expressions in the formal theory. In this code illustration, the following functionality is provided by `mathlib`: `WUnDiGraph`, `WUnDiEdge`, `Node`, `sum(., .)` and `allMappings(.)`. A demo of this code can be found in `mathlib.demos.Coherence`.

```scala
def coherence(
               network: WUnDiGraph[String],
               positiveConstraints: Set[WUnDiEdge[Node[String]]]
             ): Map[Node[String], Boolean] = {
  val negativeConstraints: Set[WUnDiEdge[Node[String]]] =
    network.edges \ positiveConstraints

  def cohPlus(assignment: Map[Node[String], Boolean]): Double = {
    def isSatisfied(pc: WUnDiEdge[Node[String]]): Double =
      if (assignment(pc.left) == assignment(pc.right)) pc.weight
      else 0.0

    sum(positiveConstraints, isSatisfied _)
  }

  def cohMinus(assignment: Map[Node[String], Boolean]): Double = {
    def isSatisfied(pc: WUnDiEdge[Node[String]]): Double =
      if (assignment(pc.left) != assignment(pc.right)) pc.weight
      else 0.0

    sum(negativeConstraints, isSatisfied _)
  }

  def coh(assignment: Map[Node[String], Boolean]): Double =
    cohPlus(assignment) + cohMinus(assignment)

  val allPossibleTruthValueAssignments =
    network.vertices.allMappings(Set(true, false))
  val optimalSolutions =
    argMax(allPossibleTruthValueAssignments, coh)
  optimalSolutions.random.get
}
```

Table 2: Mappings between formal expression and `mathlib` implementation.

Formal expression	`mathlib` implementation and description
G	`network: WUnDiGraph[String]` *Undirected weighted graph with labels.*
C^+	`positiveConstraints: Set[WUnDiEdge[Node[String]]]` *Set of positive constraints as weighted edges.*
C^-	`negativeConstraints: Set[WUnDiEdge[Node[String]]]` *Set of negative constraints, computed by subtracting the positive constraints from all edges in* `network`.
w	*Weights are represented not by an explicit function, but by a weighted graph.*
$T : V \rightarrow \{true, false\}$	`allPossibleTruthValueAssignments` *The truth value assignments are explicitly listed by generating all mappings between vertices and* `Set(true, false)`.
$Coh^+(T)$	`cohPlus(assignment: Map[Node[String], Boolean]): Double` *Returns the sum of weights of all satisfied positive constraints.*
$Coh^-(T)$	`cohMinus(assignment: Map[Node[String], Boolean]): Double` *Returns the sum of weights of all satisfied negative constraints.*
$Coh(T)$	`coh(assignment: Map[Node[String], Boolean]): Double` *Returns the sum of weights of all satisfied constraints.*
n.a.	`optimalSolutions` *Compute all truth value assignments with maximum coherence.*
n.a.	`optimalSolutions.random.get` *The formal specification is met when any maximum truth value assignment is returned, so we return a random maximum one.*

Resources

- Github repository: https://github.com/markblokpoel/mathlib
- Website: https://markblokpoel.github.io/mathlib
- Scaladoc: https://markblokpoel.github.io/mathlib/scaladoc

Acknowledgements

We thank the Computational Cognitive Science group at the Donders Center for Cognition (Nijmegen, The Netherlands) for useful discussions and feedback, in particular, Laura van de Braak, Olivia Guest, and Iris van Rooij. We also thank the reviewers Larkin Liu, Russel Richie, and Stephen Mann and the editor Daniel Katz for their useful feedback which has greatly improved this paper.

This project was supported by Netherlands Organization for Scientific Research Gravitation Grant of the Language in Interaction consortium 024.001.006, the Radboud School for Artificial Intelligence, and the Donders Institute, Donders Center for Cognition.

References

Blokpoel, M., & van Rooij, I. (2021). *Theoretical modeling for cognitive science and psychology*. https://computationalcognitivescience.github.io/lovelace/

Guest, O., & Martin, A. E. (2021). How computational modeling can force theory building in psychological science. *Perspectives on Psychological Science*, *16*. https://doi.org/10.1177/1745691620970585

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. W.H. Freeman, San Francisco, CA.

Odersky, M. (2008). *Programming in Scala*. Mountain View, California: Artima.

Thagard, P., & Verbeurgt, K. (1998). Coherence as constraint satisfaction. *Cognitive Science*, *22*(1), 24.

van Rooij, I., & Baggio, G. (2021). Theory before the test: How to build high-verisimilitude explanatory theories in psychological science. *Perspectives on Psychological Science*, *16*. https://doi.org/10.1177/1745691620970604