Mio: A Block-Based Environment for Program Design

Junya Nose
SoftBank
Tokyo, Japan
junya.nose@prg.is.titech.ac.jp

Youyou Cong
Tokyo Institute of Technology
Tokyo, Japan
cong@c.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
Tokyo, Japan
masuhara@acm.org

Abstract

Program design should be taught with a comprehensible guideline and appropriate tool support. While Felleisen et al.'s program design recipe serves as a good guideline for novice learners, no existing tool provides sufficient support for step-by-step design. We propose Mio, an environment for designing programs based on the design recipe. In Mio, the programmer uses blocks to express design artifacts, such as examples of input and output data. The system checks the consistency of the design, gives feedback to the programmer, and produces a half-completed program for use in steps after designing. A preliminary experiment in the classroom showed its ability to make program design easier for novices, and to encourage programmers to follow the design recipe. In this paper, we demonstrate the core features of Mio, report the results of the experiment, and discuss our plans for extensions.

CCS Concepts: • **Applied computing** → **Education**; • **Social and professional topics** → *Computing education.*

Keywords: program design recipe, block-based programming, pedagogic programming environment

ACM Reference Format:
Junya Nose, Youyou Cong, and Hidehiko Masuhara. 2022. Mio: A Block-Based Environment for Program Design. In *Proceedings of the 2022 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '22), December 05, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563767.3568127

1 Introduction

Problem solving through programming is a non-trivial process consisting of multiple steps. These steps are divided into *design* steps and *coding* steps. At design steps, one analyzes the problem, decides the data representation of the information involved in the problem, and sketches the structure of the computation that manipulates the data. At coding steps, one writes function definitions and confirms their correctness. The two groups of steps are related to each other, and often require reworking of earlier steps at a later stage of development.

Felleisen et al. advocate that program design should be taught along with a clear guidance, and introduced the *program design recipe* in their textbook *How to Design Programs* [5]. The design recipe allows the programmer to systematically compose programs according to the structure of input data. More concretely, the design recipe consists of six steps, among which the first four are classified as design steps:

1. Data definition and examples
2. Purpose, signature, header
3. Input-output examples
4. Template
5. Coding
6. Testing

In Figure 1, we show a Scala program that is developed by following the six steps of the design recipe. The problem here is to define a function `area` that computes the area of a given shape, which is either a square or a triangle. Among the outcomes of the intermediate steps, data examples do not directly contribute to the function definition, but creating them enhances the understanding of the problem. Similarly, templates are not visible in the complete function definition (indeed, they are not a valid expression in Scala), but building them reduces the effort required in coding. The intermediate outcomes are also useful for instructors, as they show at which step the student gets stuck.

Unfortunately, when students are asked to develop programs based on the design recipe, they do not always follow all the design steps. As an example, consider the template step. Some students may think this step is redundant and go straight to coding. Other students may try to develop a template but give up eventually because they do not know how to write it or how to check its correctness. Such skipping is observed every year in the introductory programming course taught in the authors' institution. In our experience, students who do not fully follow the design recipe can solve easier problems, but they tend to get stuck on harder problems,

```
// Data definition
sealed abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height: Double)
extends Shape

// Data examples
val square = Square(3)
val triangle = Triangle(4, 5)

// Purpose, signature, header
// Compute the area of a given shape
/*
def area(shape: Shape): Double = {
  0
}
*/

// Input-output examples
// Given square, return 9
// Given triangle, return 10
```

```
// Function template
/*
def area(shape: Shape): Double = {
  shape match {
    case Square(length) => ... length ...
    case Triangle(base, height) => ... base ... height ...
  }
}
*/

// Function definition
def area(shape: Shape): Double = {
  shape match {
    case Square(length) => length * length
    case Triangle(base, height) => base * height / 2
  }
}

// Tests
area(square) == 9
area(triangle) == 10
```

Figure 1. A Function Developed Based on Design Recipe

where the design recipe can actually be useful. A similar observation was also made by Castro [2] in their CS1 course.

Compared to requesting students to voluntarily follow the design recipe, providing a tool for describing the outcomes of the design steps would be a more promising approach to teaching program design. From such a tool, students can learn what they are expected to compose, and can receive feedback on their mistakes and potential problems. There exist several tools for assisting program design [7, 11, 12, 14, 15], but they are either insufficient for programming with complex data types, or they are focused on a particular step of the design recipe.

In this paper, we present Mio, an environment for designing programs based on the design recipe. Mio has the following features.

- It provides a block-based syntax for design steps. This reduces the cost of learning "languages for design", which may be complex (as in the case of data definitions) or not be part of the programming language for coding (as in the case of templates).
- It displays the code representation of blocks on the fly. This allows the programmer to see how each step contributes to the eventual program.
- It generates feedback on the outcomes of design steps. This reduces common mistakes such as non-exhaustive examples and pattern matching.

Note that Mio is in an early stage of development. Its support is currently limited to non-recursive data types (although the idea scales to recursive ones) and its effectiveness has not yet been assessed. However, an experimental use of Mio in the classroom shows its potential effectiveness, giving us confidence that the overall design is on the right track.

In the rest of this paper, we demonstrate the functionalities of Mio (Section 2) and report the results of our preliminary experiment (Section 3). We then discuss the design of Mio from two points of view (Section 4) and describe three extensions we plan to implement (Section 5). Lastly, we compare our work with related studies (Section 6) and conclude the paper with future perspectives (Section 7).

2 A Walk Through of Mio

2.1 Overview

Mio is a browser-based environment built on top of Google Blockly[1]. It consists of a block palette, a workspace, and an editor. The programmer goes through the four design steps one by one, and at each step, they use blocks to compose the required elements while seeing their code representation.

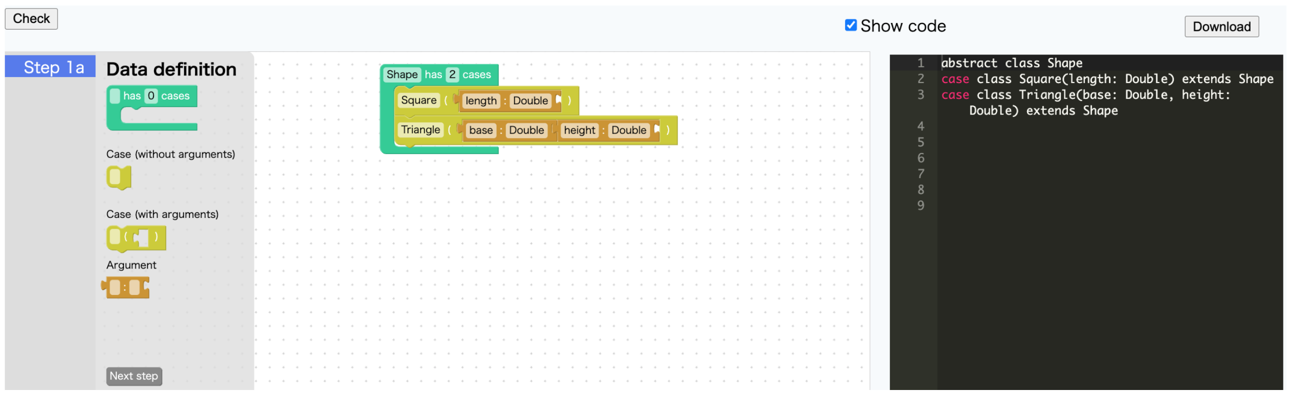
Below, we walk the reader through the four design steps, and describe the consistency checking feature. Throughout the section, we use the `area` function to illustrate the tasks of each step.

2.2 The Design Steps

Step 1a: Data Definition. In the first half[2] of Step 1, we define shapes as an algebraic data type (Figure 2). We drag the C-shaped data definition block into the workspace, and write the name of the data type and the number of cases. We

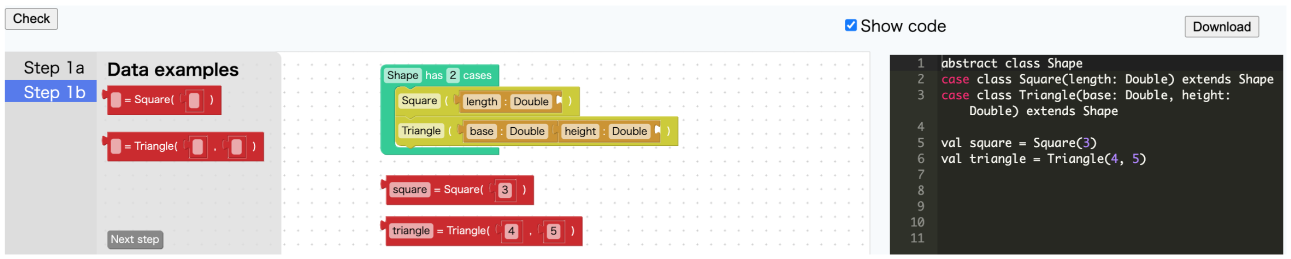
[1] https://developers.google.com/blockly

[2] The idea of splitting Step 1 into two is borrowed from Ramsey [10].



```
abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height:
    Double) extends Shape
```

Figure 2. Step 1a



```
abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height:
    Double) extends Shape

val square = Square(3)
val triangle = Triangle(4, 5)
```

Figure 3. Step 1b

then plug two case blocks into the hole of the data definition block, and specify the names of constructors as well as their arguments. Note that there are two kinds of case blocks: one with arguments and the other without. The distinction is used to guide the conversion into code (`case class` vs. `case object` in the case of Scala) and the creation of data examples (see next step). In the editor on the right, we see the Scala data definition converted from the block data definition.

When we complete data definition, we click the "Next step" button to ask Mio for the permission to proceed to the next design step. If approved, we see a new item "Step 1b" in the block menu bar.

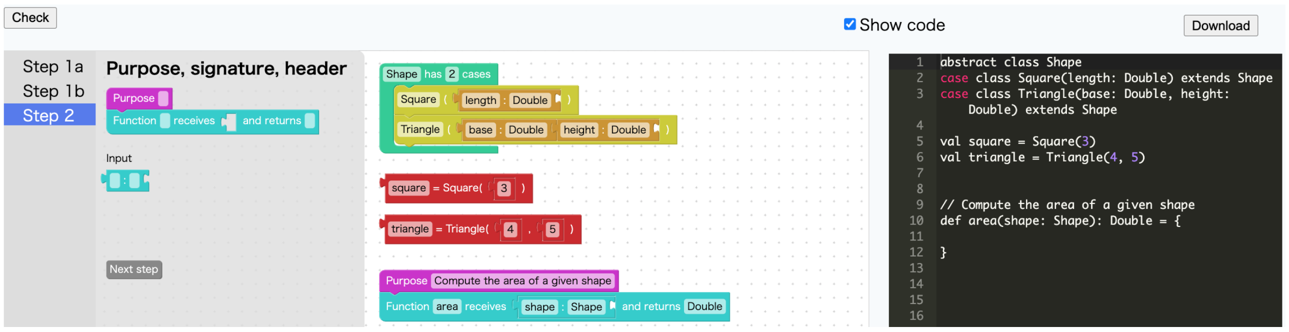
Step 1b: Data Examples. In the second half of Step 1, we create examples of shapes (Figure 3). When we proceed to this step, Mio automatically generates two kinds of blocks: one for creating squares and the other for creating triangles. Using these blocks, we create at least one example for each case. Observe that the blocks allow us to assign each data example a name, which is to be used in input-output examples. For constructors with no arguments, Mio generates a simpler block that does not have the name field (as it is not necessary). And again, we can immediately see the Scala version of the data examples.

Step 2: Purpose, Signature, Header. As the second step, we compose a purpose statement, write a signature, and create a function header (Figure 4). We do this by filling in the holes of the function header block. In the editor, we see a stub (called "header" in *How to Design Programs*) of the `area` function in the Scala syntax.

Step 3: Input-output Examples. Having obtained a header, we create input-output examples of the `area` function (Figure 5). Here we use the two-hole block as well as the data examples from Step 1b, which automatically show up in the block palette. On the right, we see two equations converted from the example blocks, serving as the tests for the function.

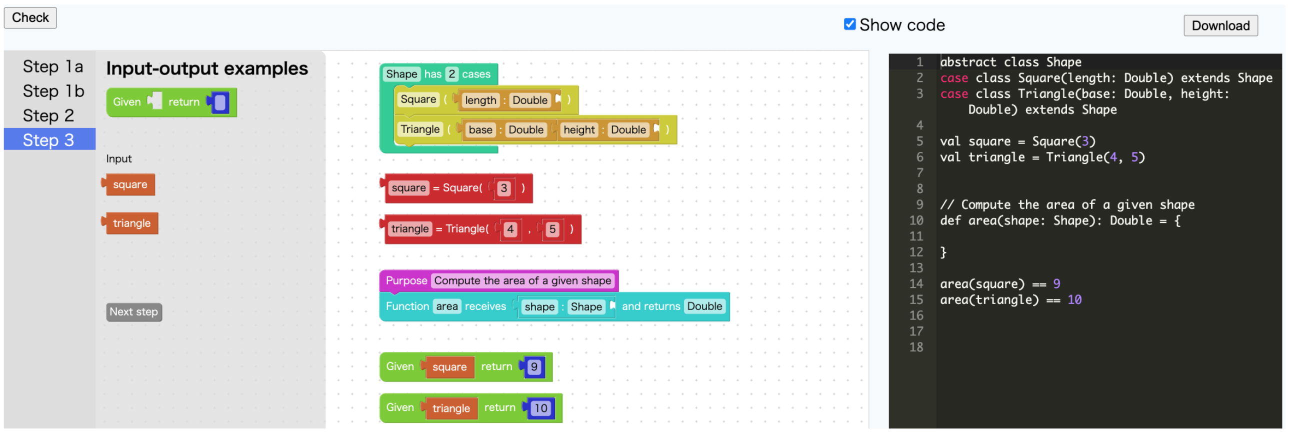
Step 4: Template. As the last step of program design, we develop a function template (Figure 6). A template is the outline of the function, and can be systematically derived from the definition of the input data. In our case, we are dealing with a data type that has two cases, hence we build a pattern matching that has two clauses. Furthermore, we are likely to use the constructor arguments in the two clauses, therefore we use the hint blocks to remember this. Similar to previous steps, we obtain a Scala pattern matching with hints on the right-hand side of case clauses.

Step 5 & 6: Coding and Testing. After Step 4, we switch to text-based programming and work on the remaining tasks. More specifically, we complete the function definition by filling in the "..." in the template, and check the behavior of the function by running the tests converted from input-output examples. In the current implementation of Mio, we cannot run programs within the environment, hence we need


Check
Show code
Download
Step 1a
Step 1b
Step 2
Purpose, signature, header
Purpose
Function receives and returns
Input
Next step
Shape has 2 cases
Square length Double
Triangle base Double height Double
square = Square(3)
triangle = Triangle(4 , 5)
Purpose Compute the area of a given shape
Function area receives shape : Shape and returns Double

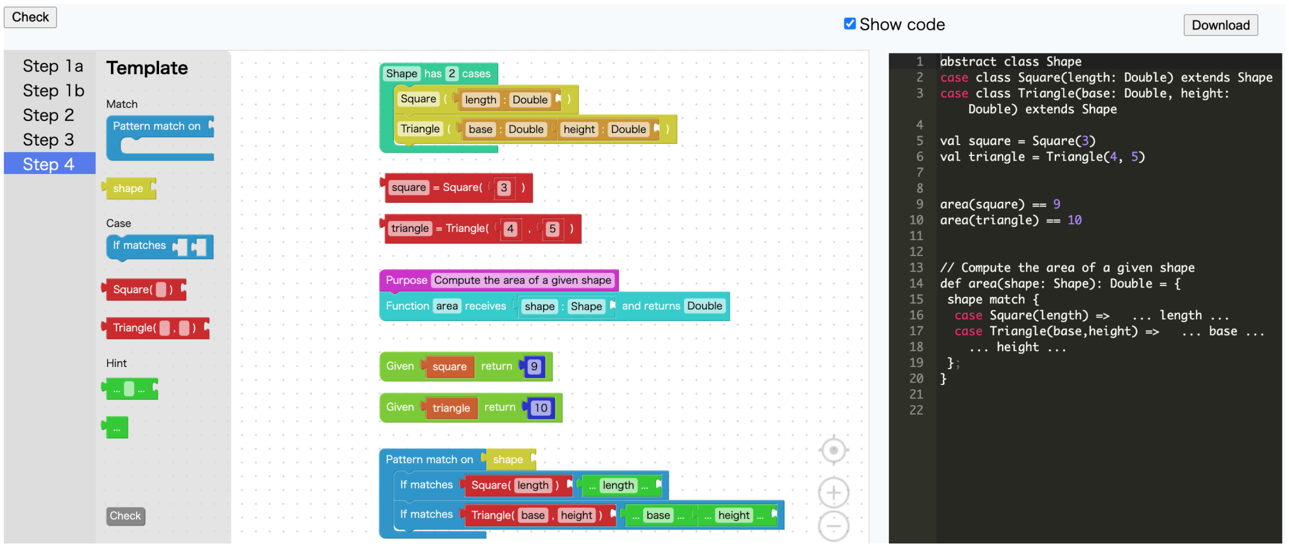
```
abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height:
    Double) extends Shape

val square = Square(3)
val triangle = Triangle(4, 5)


// Compute the area of a given shape
def area(shape: Shape): Double = {

}
```

Figure 4. Step 2



```
abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height:
    Double) extends Shape

val square = Square(3)
val triangle = Triangle(4, 5)


// Compute the area of a given shape
def area(shape: Shape): Double = {

}

area(square) == 9
area(triangle) == 10
```

Figure 5. Step 3



```
abstract class Shape
case class Square(length: Double) extends Shape
case class Triangle(base: Double, height:
    Double) extends Shape

val square = Square(3)
val triangle = Triangle(4, 5)


area(square) == 9
area(triangle) == 10


// Compute the area of a given shape
def area(shape: Shape): Double = {
 shape match {
  case Square(length) =>   ... length ...
  case Triangle(base,height) =>   ... base ...
    ... height ...
 };
}
```

Figure 6. Step 4

to download the code and run it using an installed Scala interpreter or some external browser-based environment (e.g., Scastie[3]).

[3]https://scastie.scala-lang.org/

2.3 Consistency Checking

When designing a program in Mio, the programmer can receive feedback on the intermediate outcomes by pressing the "Check" button or the "Next step" button. To see what

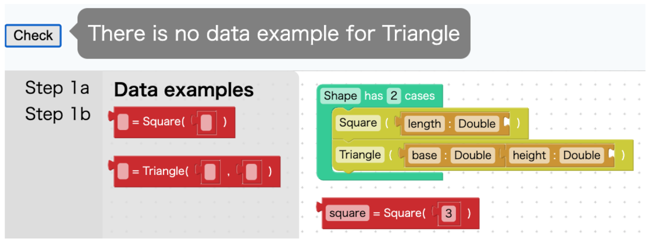

Check
There is no data example for Triangle
Step 1a
Step 1b
Data examples
= Square(
= Triangle(
Shape has 2 cases
Square (length : Double)
Triangle (base : Double height : Double)
square = Square(3)

Figure 7. Warning on Inssufficient Data Example

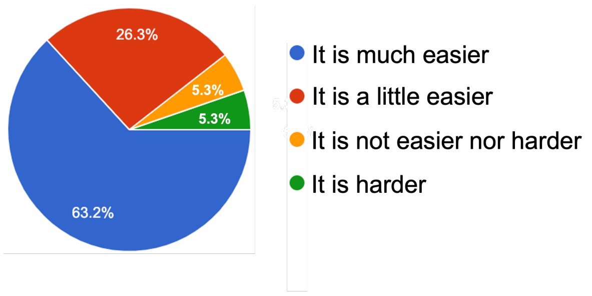


Figure 8. Responses to Q1 ($N = 19$)

kind of feedback Mio generates, suppose the programmer created an example of squares but did not create an example of triangles. In this case, Mio warns the programmer that the data example is insufficient (Figure 7). Suppose next the programmer developed a template with a case clause for `Square` but without one for `Triangle`. In this case, Mio complains that the number of clauses is wrong. Thus, Mio guarantees the consistency of data definitions, data examples, and function templates.

When the programmer goes back to an earlier step and makes changes, they can easily see necessary changes for subsequent steps. Consider the case where the programmer added a case `Circle(radius: Double)` to the `Shape` data type after creating data examples. In such a case, Mio generates a block for creating `Circle` and displays it in the block palette when the programmer revisits Step 1b. Consider next the case where the programmer added an argument `flag` to the `area` function after creating input-output examples. In such a case, Mio inserts an additional hole into the existing input-output example blocks when the programmer revisits Step 3. Thus, Mio maintains consistency of intermediate outcomes in back-and-forth development.

After fixing such mistakes, the programmer sees a pop-up message of the form "Step N OK". Then, they can proceed to the next step with confidence.

3 Preliminary Experiment

Although Mio is still under development, we have conducted a small experiment to solicit feedback from students. The participants of the experiment are 19 undergraduate students enrolled in "Introduction to Computer Science", a functional programming course taught by the second author using the Scala language. The course introduces the design recipe on day 1, and each lesson starts with step-by-step design of a sample program involving a new data type. The experiment was done during the last lesson of the course; at this point, the students had 6-week experience in programming with non-recursive and recursive data types.

In the experiment, we gave a brief introduction to Mio through a video demonstration created by the first author, and asked the students to develop a template of the `area` function in (a Japanese version of) Mio. We instructed the students to submit a screenshot of the final outcome, together with the answers to the following questions.

Q1. Do you think programming in Mio is easier than programming in a standard editor?
Q2. What did you like about Mio, and what did you think could be improved?

In Figure 8, we present the responses to the first question. The responses are mostly positive, meaning that the students generally appreciate the support provided by Mio. As the responses to the second question, we received the following comments.

- I found Mio useful because I don't need to worry about the syntax and grammatical errors.
- I liked the colored blocks. It was also fun to see how the code grows.
- I found it convenient to be able to reuse data examples in input-output examples.
- I felt encouraged when Mio gave me positive feedback.
- I think coding in text is easier when you have some experience.

From these comments, we conclude that the use of blocks in program design could be beneficial to novice programmers, and showing their code representation and providing feedback would be effective in motivating programmers to follow design steps. We also learned that allowing text-based editing is important for making Mio usable to a wider range of programmers.

4 Discussion

4.1 Learning Costs

As we saw in Section 2, Mio uses blocks and text for different phases of programming. The reader may wonder whether this would incur additional learning cost, or, put differently, whether it would be more natural to use a uniform interface for the entire process. We think that the use of two interfaces is reasonable. In the design phase, it is easier to work with blocks as the outcomes in this phase involve non-trivial syntax and language-specific keywords. In the coding phase, it is more convenient to work with text because the code to be written in this phase is often simple (consisting mainly of function calls, primitive operations, and constructors). We hope to justify these points in our future experiments.

4.2 Scalability

In Section 2, we showed how Mio helps the programmer design a simple program written in a functional language. This does not mean that Mio is only useful for teaching introductory functional programming. The underlying design recipe is a guidance for problem solving in general, and hence, the environment can be adjusted to more complex problems and other programming paradigms. In particular, it should be straightforward to create a variation of Mio for teaching object-oriented programming. The idea is to follow the design recipe for classes and objects [6], which has similar steps to the design recipe we have been using so far. It would be interesting to build this variation of Mio and investigate its effectiveness in an object-oriented programming course.

5 Extensions

In the current version of Mio, we support non-recursive data types and generation of Scala code, and we require all the tasks to be done using blocks. As future work, we intend to extend Mio with various forms of recursion and code generation in other languages, as well as a mechanism for allowing text-based editing. Below, we briefly describe our plans for these extensions.

5.1 Variations of Recursion

Recursion is known as one of the most challenging concepts taught in an introductory programming course. The design recipe simplifies designing of recursive functions by drawing the programmer's attention to the structure of input data. For instance, in the case of structural recursion, the design recipe asks the programmer to identify the recursive arguments of constructors when defining data, and to insert recursive calls on those arguments when building a template. With these recursive calls at hand, it is often trivial to complete the function definition. Moreover, the function is guaranteed to terminate, as every recursive call is made with a structurally smaller argument. We plan to support structural recursion by providing special blocks for recursive constructor arguments and hints representing recursive calls.

In *How to Design Programs*, there are also design recipes for more advanced forms of recursion, including mutual recursion, accumulative recursion, and generative recursion (as found in divide-and-conquer algorithms). Among these, mutual recursion requires simultaneous definition of multiple data types. Accumulative recursion gives rise to a helper function, which takes in an accumulator argument, and a call to the helper function, which sets the accumulator to its initial value. Generative recursion is slightly trickier. The function definition is no longer driven by the structure of input data; instead, it deals with the trivial and non-trivial cases of input data, which are classified according to the given problem. To support these variations, we plan to provide different modes of recursive program design and ask the programmer to pick one before starting the design process. We would also like to support switching from one mode to another mode by performing a consistency checking similar to what we do in back-and-forth development.

5.2 Code Generation in Other Languages

As can be seen from the screenshots in Section 2, blocks in Mio are designed in a language-neutral way. In other words, they do not have any elements that are specific to the Scala language. For this reason, it is easy to convert blocks into code in other functional languages, such as Haskell, OCaml, and Racket. Note that, although Racket is equipped with structures instead of algebraic data types, there is a rigorous correspondence between the two. For instance, extracting a constructor argument is equivalent to applying a selector function, and pattern matching is the same as conditionals with tests of the form `(datatype-name? x)`.

5.3 Mixed Use of Blocks and Text

Programming with blocks is attractive to beginners, but as the participants of our experiment pointed out, it would be verbose for experienced programmers. To reduce the burden caused by blocks, we are planning on allowing text-based editing by implementing a translation from code to blocks. It is however not obvious how we could implement such a translation: code is much less structured than blocks, and the programmer may write any expression in any place. This suggests that we would need to restrict code editing in some way, and we are currently trying to find appropriate restrictions.

6 Related Work

Hybrid Programming Environments. In response to the increasing diversity of the computing population, researchers have developed a variety of block-based programming environments over the past decades. Among these environments, those called "hybrid" offer a text interface in addition to a block interface. A notable example of hybrid environments is BlockPy [1], which allows the programmer to switch between blocks and text at any time. Another example is Pencil.cc [13], whose hybrid mode allows the programmer to drag blocks into the text editor.

Program Design Using Blocks. While blocks are widely used to assist coding, they are rarely used to assist designing. The only exception we are aware of is the work by Rivera et al. [11], who create a variation of the Snap*!* [8] programming environment that helps plan composition with higher-order function blocks, such as `map` and `filter`. Like us, they use blocks solely for designing, but unlike us, they do not guide step-by-step development of programs.

Environments Based on Design Recipe. There are several programming environments that encourage the programmer to follow a subset of the steps of the design recipe. WeScheme [15] is a browser-based environment for the Racket language. It asks the programmer to provide the signature and input-output examples before defining functions. DRaCO [12] is a similar environment, and it additionally asks for a purpose statement and the function's effects. Mio extends these environments with the principle of data-driven development, which allows systematic design of programs involving complex data types.

Support for Specific Design Steps. There also exist tools dedicated to a particular step of the design recipe. D4 [7] is a system for teaching how to organize data. Using D4, the programmer can see the consequences of data organization through a series of exercises, such as creating a data example and extracting a value from data. Examplar [14] is an IDE for teaching how to write tests. In Examplar, the programmer can assess the quality of their tests by running them against correct and wrong implementations provided by the instructor. We are interested in enriching Mio's functionalities by borrowing ideas from these studies.

Design Recipe for Program Synthesis. The design recipe not only helps human write programs, but also helps computers synthesize programs. Levine and Tobin-Hochstadt [9] generate function bodies using signatures and input-output examples. Feldman et al. [3] find complete function definitions using the programmer's incomplete definitions (which can be viewed as a generalization of templates). These tools are both developed for educational purposes, such as providing feedback and suggesting test cases. We conjecture that having the outcomes of all design steps would be effective in improving the performance of synthesis.

7 Conclusion

Teaching program design is not just about giving a set of steps to follow; it is also about motivating students to follow the steps. We address the latter challenge by building an environment that integrates Felleisen et al.'s guidance. We believe that the main features of our environment—namely the block interface, code rendering, and feedback generation—would contribute to better learning experience and outcomes.

After implementing the extensions discussed in Section 5, we intend to conduct a long-term study on the effectiveness of our environment. We are particularly interested in the impact on the rate of correct solutions to the course assignments and the students' programming habits in the subsequent programming courses. In fact, Felleisen et al. [4] showed that the design recipe has positive effects on both aspects. They derived this conclusion by comparing students who learned and did not learn the design recipe. We hope to obtain similar results when comparing students who learned the design recipe with and without our environment.

Acknowledgments

We are grateful to the anonymous reviewers for their thorough reading and insightful comments. We would also like to thank the members of the Programming Research Group at Tokyo Tech and the participants of Dagstuhl Seminar No. 22302 for their encouragement and suggestions. This research was partly supported by JSPS KAKENHI Grant Number JP20K21790.

References

[1] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A Shaffer, and Dennis Kafura. 2017. BlockPy: An open access data-science environment for introductory programmers. *Computer* 50, 5 (2017), 18–26.

[2] Francisco Enrique Vicente G Castro. 2020. *Development of a Data-Grounded Theory of Program Design in HTDP.* Ph. D. Dissertation. Ph.D. Dissertation. Worcester Polytechnic Institute. https://digitalcommons.wpi.edu/etd-dissertations/595

[3] Molly Q Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards Answering "Am I on the Right Track?" Automatically Using Program Synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3358711.3361626

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The structure and interpretation of the computer science curriculums. *Journal of Functional Programming* 14, 4 (2004), 365–378. https://doi.org/10.1017/S0956796804005076

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press.

[6] Matthias Felleisen, Matthew Flatt, Robert Bruce Findler, Kathryn E. Gray, Shriram Krishnamurthi, and Viera K. Proulx. 2012. How to Design Classes - Data: Structure and Organization. https://felleisen.org/matthias/HtDC/htdc.pdf

[7] Xingjian Gu, Max A. Heller, Stella Li, Yanyan Ren, Kathi Fisler, and Shriram Krishnamurthi. 2020. Using Design Alternatives to Learn About Data Organizations. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 248–258. https://doi.org/10.1145/3372782.3406267

[8] Brian Harvey, Daniel Garcia, Josh Paley, and Luke Segars. 2012. Snap! (Build Your Own Blocks) (Abstract Only). In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 662. https://doi.org/10.1145/2157136.2157351

[9] Hazel Levine and Sam Tobin-Hochstadt. 2022. Automating the Design Recipe. Presented at the Scheme and Functional Programming Workshop (Scheme '22).

[10] Norman Ramsey. 2014. On Teaching *How to Design Programs*: Observations from a Newcomer. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 153–166. https://doi.org/10.1145/2628136.2628137

[11] Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*. Association for Computing Machinery, 84–104.

[12] Mike Dongyub Ryu. 2018. *Improving Introductory Computer Science Education with DRaCO.* Master's thesis. California Polytechnic State University.

[13] David Weintrop and Uri Wilensky. 2018. How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction* 17 (2018), 83–92.

[14] John Wrenn and Shriram Krishnamurthi. 2019. Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, 131–139.

[15] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2011. WeScheme: the browser is your programming environment. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. Association for Computing Machinery, 163–167.