JFP **35**, e12, 28 pages, 2025.

doi:10.1017/S0956796825000073

OCaml Blockly

KENICHI ASAI

Department of Information Science, Ochanomizu University, Tokyo, Japan
(*e-mail:* `asai@is.ocha.ac.jp`)

Abstract

OCaml Blockly is a block-based programming environment for a subset of the functional language OCaml, developed based on Google Blockly. The distinct feature of OCaml Blockly is that it knows the scoping and typing rules of OCaml. As such, for any complete program in OCaml Blockly, its OCaml counterpart compiles: it is free from syntax errors, scoping errors, and type errors. OCaml Blockly supports introductory constructs of OCaml that are sufficient to write the shortest path problem for the Tokyo metro network. This paper describes the design of OCaml Blockly and how it is used in a CS-major course on functional programming.

1 Introduction

A block-based programming environment serves as an attractive introduction to programming for all kinds of people. The wide adoption of block-based programming environments is due in part to easy installation (or no installation because it runs on a browser), fun contents (one can program animation quickly), and community (created games can be shared among friends). Apart from these nontechnical aspects, a block-based programming environment offers technical benefits in programming: it avoids syntax errors, which is one of the initial obstacles most beginners face.

The benefits of syntax support are not restricted to beginners. CS-major students do suffer from syntax problems when they first start programming. Even if we expect them to be able to handle syntax problems eventually, having syntax support at the beginning could help them concentrate on the more important parts of programming.

However, most of the block-based programming environments so far are for imperative languages. In particular, they do not support lexical scoping or sophisticated type checking that are common in typed functional languages. Without these features, even if a constructed program is free from syntax errors, it does not compile in general: it could have scoping errors or type errors.

In this paper, we present OCaml Blockly, a variant of Google Blockly. On top of the standard syntax support, OCaml Blockly performs scope checking at program construction time: a variable block can appear only in its scope. Similarly, OCaml Blockly performs type checking at program construction time: only a well-typed program can be built. As a result, once a program is constructed in OCaml Blockly, it compiles. This paper first presents the design of OCaml Blockly and discusses its merits as well as its limitations.

OCaml Blockly has been used in a functional programming course for CS-major students in Ochanomizu University since 2019. This paper describes how OCaml Blockly was perceived by students. In particular, we report that the block-based programming is useful for the basic parts of the course and that migration to text-based programming does not appear to be a serious problem in our setting.

The contributions of this paper are summarized as follows:

- We present a design of a block-based programming environment for a functional language where scoping and typing rules are enforced at program construction time.
- We have implemented OCaml Blockly that realizes these features and discuss its merits as well as limitations.
- We report on the use of OCaml Blockly in a CS-major functional programming course and summarize students' evaluation.

The paper presents the OCaml Blockly language in the next section and continues with students' typical interaction with OCaml Blockly in Section 3. The functional programming course is described in Section 4 to which OCaml Blockly is integrated, followed by how OCaml Blockly is used in a classroom and students' evaluation in Section 5. Section 6 discusses related work and the paper concludes in Section 7.

2 The OCaml Blockly language

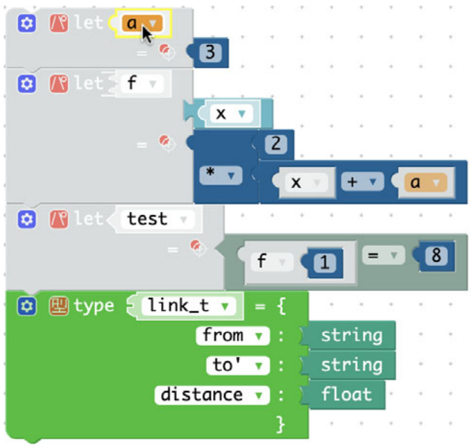
In this section, we describe the OCaml Blockly language. It is a limited subset of OCaml that has constructs commonly taught in an introductory functional programming course. It supports variables, recursive functions, conditionals, user-defined records, lists, pattern matching, and higher-order functions, but not mutual recursion, variants, exceptions, modules, side effects, and mutations.

OCaml Blockly is implemented by extending Google Blockly, a visual programming editor for various imperative languages. Since Google Blockly is an extensible system in which we can add our own blocks, we implemented all the basic constructs of OCaml.[1] However, there are mainly three aspects that require thorough consideration: lexical scoping, type checking, and higher-order functions. Below, we first introduce the basic structure of blocks. We then discuss the design of these three critical features.

2.1 Structure of blocks

OCaml Blockly inherits the block structure of Google Blockly. There are two types of blocks: statement blocks and expression blocks. A statement block has connectors at the top and the bottom of the block. It is used to represent a declaration in OCaml Blockly, such as variable and function definition (a `let` block without `in`) and type declaration (a `type` block). For example, in Figure 1, a variable `a` is first defined and then used to define a function `f`. The variable `test` records whether `f 1` evaluates to `8`. Finally, a type `link_t` is defined as a record type that has three fields: two strings and a number.

[1] We have implemented all the constructs manually, rather than taking more general approach such as using the language server protocol.



Equivalent OCaml program:

```
let a = 3

let f x = 2 * (x + a)


let test = f 1 = 8


type link_t = {
  from     : string;
  to'      : string;
  distance : float;
}
```

Fig. 1. Variable, function, and type declaration.

An expression block has a connector at the left of the block and optionally receives its arguments either in holes (inline arguments) or from the connector at the right of the block (external arguments). In Figure 1, all the blocks other than `let` and `type` blocks are expression blocks. Among them, + receives its arguments inline while ∗ receives its arguments externally. As a whole, the function body represents `2 * (x + a)`. One can switch between inline/external arguments by right-clicking the block and selecting an appropriate option from the dropdown menu.

Each expression block has a type, reflected in the shape of its connector. An OCaml Blockly type may not only be a first-order type, such as `int` (the connector of numbers in Figure 1) or `bool` (the output connector of =), but it may also be a higher-order function type. The complex connector shape as observed in the `let` block for `f` in Figure 1 represents a function type consisting of its parameter and return types (in this case, both `int`). Note that when we use the function, as in the `test` block below, its output connector has the shape of the return type only, because the block receives its argument in the hole. The complex shape appears only when the function is used as an argument of a higher-order function (see Section 2.4).

Some blocks have a few dialog buttons on them. In Figure 1, the first blue button on `let` blocks configures the number of function parameters, while the one for the `type` block configures the number of record fields.

The second red button on the `let` blocks depicts the first Japanese Katakana character for “pattern” and lists possible patterns for parameters, such as variables, tuples, and previously defined records. Similarly, the second red button on the `type` block depicts a Chinese character for “type” and lists all the available types to be used for specifying types of fields.

Finally, by right-clicking a block, we can see more operations on the block, such as toggling arguments between external and inline, adding a `rec` keyword to `let` blocks, and folding blocks. Folding of blocks is particularly useful when building a large program.

As for all block-based programming environments, nesting structure of blocks naturally represents nesting structure of expressions. Since Blockly knows the operator precedence, parentheses are automatically inserted in the textual program when a low-precedence operator appears as an argument of a high-precedence operator.

2.2 Scope of variables, functions, and type declarations

As in OCaml, a variable or function definition has a scope: defined variables and functions can be used only at blocks that are connected below their definitions. In Figure 1, the use of `f` in `test` is legal, because the `test` block is connected below the definition of `f`. The same goes for parameters. The variable block `x` can be used only in the body of `f` (at the connector to the right of the = sign).

Variable blocks are created when a variable name in a `let` block is dragged. They can be dropped to a legal place, i.e., a place where the variable can be used in a scope- and type-safe manner. While a variable block is being dragged, the environment shows a warning saying that it cannot be dropped there. If a variable block is moved to a legal place, the warning disappears and the variable block can be dropped there. If it is dropped to an illegal place, it is automatically deleted.

To enforce the scoping rule of OCaml, OCaml Blockly maintains connections between definitions and uses of variables and functions. For example, when the user places the cursor at the variable `a` as in Figure 1, all the occurrences of `a` are highlighted. The user can also perform α-renaming of variables by clicking one of the variable blocks, selecting "Rename variable..." from the menu, and typing a new variable name. Illegal variable names are rejected. For instance, if one uses the name `to` (which is a reserved keyword in OCaml) instead of `to'` for the second field of `link_t`, the environment tells the user that the name is invalid. Illegal cases include name clashes. As an example, if one tries to rename the variable `a` in Figure 1 to `x`, the environment yields an invalid name error, because the variable `x` is declared in the body of `f` and `a` is used within the scope of `x`.

The scoping rule applies to type declarations, too. A record block, which can be created by dragging the record name, can exist only below the declaration block. The α-renaming of field names is also supported.

2.3 Type checking

Since OCaml is a statically typed language with Hindley-Milner type inference (Pottier & Rémy, 2004), all the blocks in OCaml Blockly are typed. The type information of a block is reflected in the shape of its connectors. Each base type has its own shape: a boolean-typed block has a triangle connector, whereas a string-typed block has a Bézier-curved hump connector. A structured type has a complex shape composed of the shapes of its constituent types. The shape of a pair type consists of the shapes of its two element types aligned vertically; the shape of `'a list` consists of the shape of `'a` with a bar below; and the shape of a function type consists of the shapes of its parameter types and return type.

OCaml Blockly supports the standard let-polymorphic type inference (Pierce, 2002). A polymorphic type has a general square-shaped connector with a color. In Figure 2, for example, the identity function `id` is assigned a polymorphic type. The two uses of `id` below show two different colors for `id`, expressing that it can be used for arguments of different types.

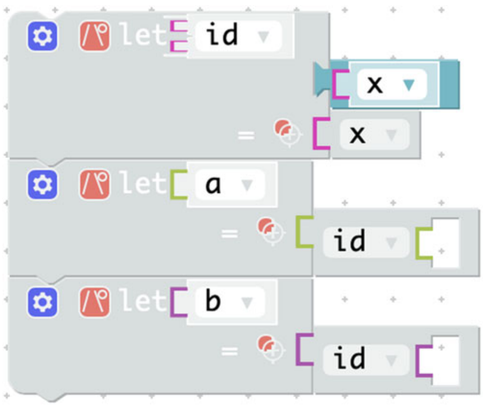


Fig. 2. Polymorphic type checking for an identity function.

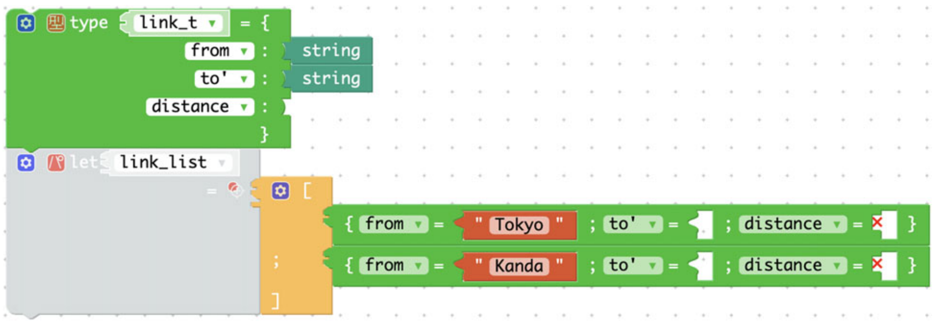

type link_t = {
from : string
to' : string
distance :
}
let link_list
= [
{ from = " Tokyo " ; to' = ; distance = }
;
{ from = " Kanda " ; to' = ; distance = }
]

Fig. 3. Connector is disabled when its type is not defined.

Type inference runs for every user action. During type inference, unconnected connectors are assumed to be of any type that satisfies the constraints coming from other blocks connected to them. When one tries to connect a block to a connector, type inference runs and checks if the types of the block and the connector can be unified. If not, a type error message appears as a pop-up message.

Since type information is necessary to connect blocks, one has to first specify types of fields in a record declaration before fields of a concrete record can be filled with blocks. Figure 3 shows a record declaration where the type of the third field is not specified yet. Accordingly, the connectors for the third field in the records below indicate that they are not active for connection at this point. If the block specifying the type of a field is removed, the corresponding connectors are disabled, resulting in the removal of blocks connected to that field. For example, if one removes `string` block for the first field in Figure 3, the two string blocks `"Tokyo"` and `"Kanda"` are removed because the connectors are disabled.

2.4 Higher-order functions

By dragging the name of a variable definition, one can construct a variable block. Similarly, by dragging the name of a function definition, one can construct a function call block

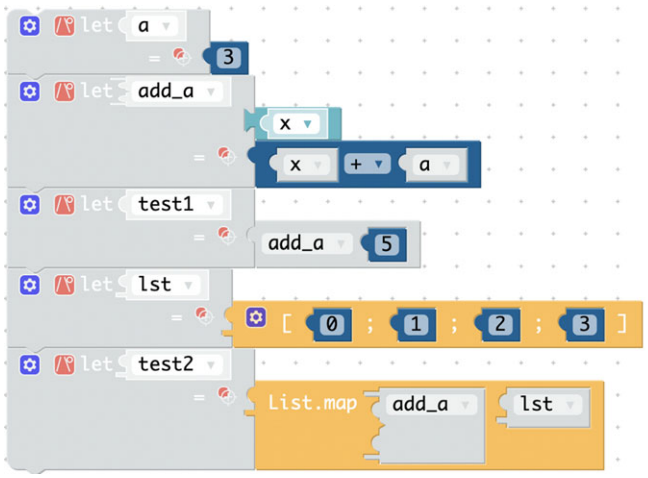


Fig. 4. Passing `add_a` to `List.map`.

having a hole for each of its arguments. Having holes for arguments is a nice default behavior for a function block. For the case when a function is used as a higher-order value, OCaml Blockly provides a way to create a function block without holes by option-dragging the function name, i.e., dragging the function name while pressing the option key.

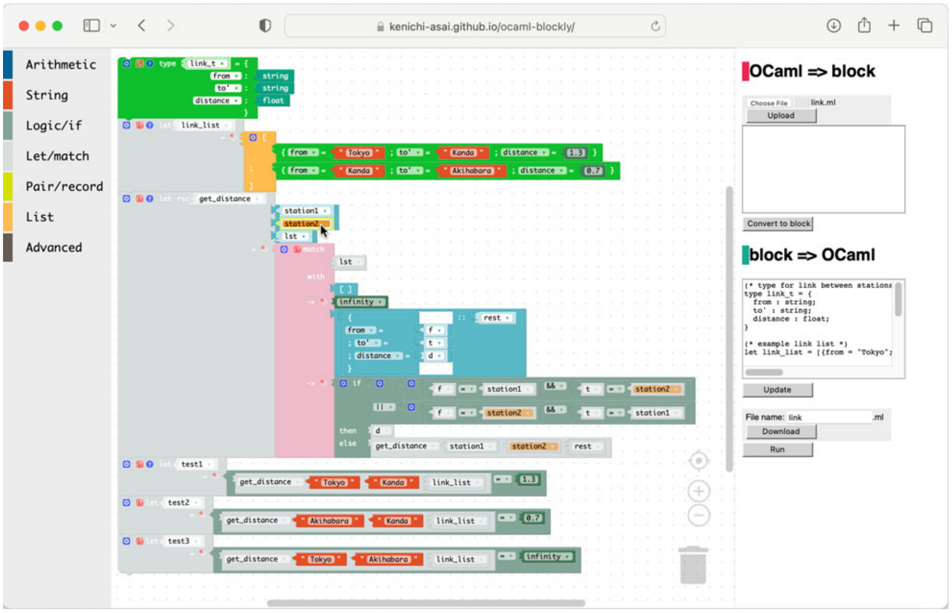
For example, at `test2` in Figure 4, the function `add_a` is mapped to a list of integers. Here, the function block `add_a` without a hole is passed to `List.map`. Notice that the complex connector of `add_a` shows a function type: it receives an integer and returns an integer. In contrast, when a function receives arguments in holes, the outermost connector shows only the type of the return value. For example, at `test1` in Figure 4, the connector for `add_a` with a hole indicates that it returns an integer, given an integer as an argument in the hole.

When one wishes to apply a function partially, one can add or remove holes by right-clicking the block and selecting an appropriate option in the menu. The shape of the output connector of the function changes as the number of holes changes.

3 Ergonomics of OCaml Blockly

In the previous section, we covered the basic design of OCaml Blockly. In this section, we show how students interact with OCaml Blockly.

Figure 5 shows an overview of the OCaml Blockly environment, opened in a browser. It is divided into three parts. The left part lists menus for various kinds of blocks roughly categorized by their types. The middle part is the main playground where one constructs a program. For example, Figure 5 shows a linear search program that returns distance between the two adjacent stations. To construct a program, one opens one of the menus in the left part and drag and drop a block (Figure 6). The right part of Figure 5 has file input/output and program execution. Apart from constructing a program from scratch using blocks, one can either upload a file or write a textual program in the textbox under “OCaml => block”. The textbox is mainly used for importing a program created in the design recipe page (see Section 4.2).



```
(* type for link between stations
type link_t = {
  from : string;
  to' : string;
  distance : float;
}

(* example link list *)
let link_list = [{from = "Tokyo";
```

Fig. 5. OCaml Blockly.

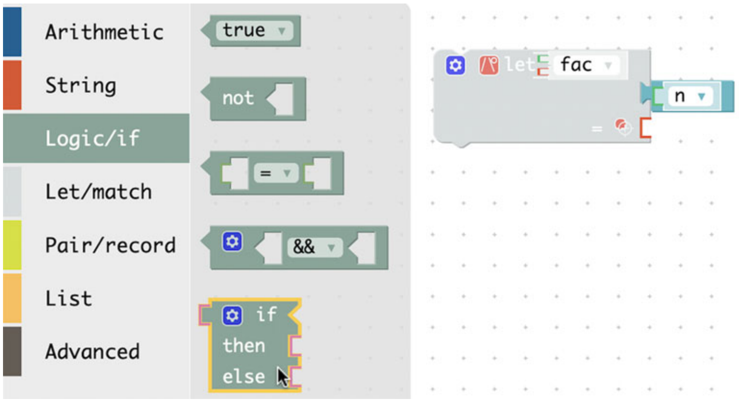


Fig. 6. Opening a menu to drag and drop a block.

When one constructs a program, its textual representation appears in the second textbox in the right part. (In the current implementation, certain kinds of changes require manual update action via the update button.) The constructed program can be downloaded as a `.ml` file.

When a complete program is constructed, it can be executed by pressing the Run button. A new window shows up and the textual result of execution is displayed. Program execution is realized by running the OCaml toplevel at the client side compiled via the js_of_ocaml compiler (Vouillon & Balat, 2014) and by sending the textual representation of the user program to it.

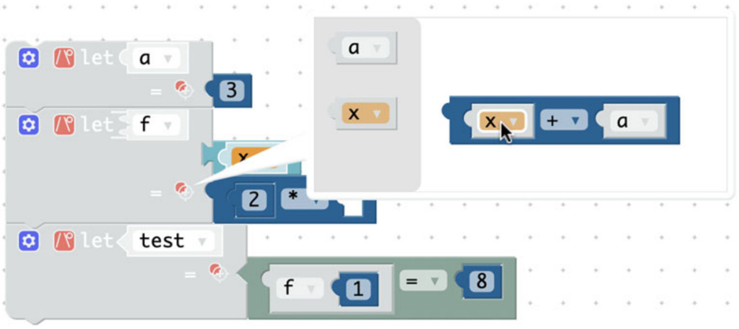


Fig. 7. Scope playground.

Like any block-based programming environments, OCaml Blockly offers easy construction and editing of code elements. Apart from the standard benefits, OCaml Blockly offers features that are specific to typed functional languages introduced in Section 2. In the following sections, we describe how those features affect user interaction as well as what limitations OCaml Blockly has.

3.1 Scope playground

OCaml Blockly enforces that a variable can be placed only within its scope. This enforcement of the scoping rule impacts how users interact with the system. There are two ways to compose blocks: top down and bottom up. In the top-down approach, one places the outermost block first. For example, to create the expression `2 * (x + a)`, one first places the `*` block, and then constructs its arguments. This interaction is always possible in the main playground. However, one would also wish to compose blocks in a bottom-up way, by first constructing `x + a` and gradually forming the whole blocks by composing smaller blocks. Such an interaction is impossible at the main playground, because one cannot place the variables `x` and `a` in that area. To recover bottom-up composition of blocks, OCaml Blockly offers a *scope playground* as shown in Figure 7.

A scope playground is a place where valid variables at that point may appear on their own. A scope playground is opened by clicking the scope button to the left of the body of a `let` block (or each branch of a `match` block). A scope playground is the same as the main playground except that variables (and records) in the scope may appear freely. For example, in the scope playground in Figure 7, one can use `x` and `a`, which appear in the menu of the scope playground. (If `f` were defined using `let rec`, `f` would have appeared in the scope playground, too.) Once blocks are constructed in a scope playground, they can be dragged to the main playground. The available variable blocks differ according to the place of a scope playground. For example, in the scope playground for `test` in Figure 7, one can use `a` and `f`.

The scope playground is also a good place to keep blocks with free variables temporarily. Suppose a user constructed the following function definition:

```
let a = 3
let f x = x + a
```

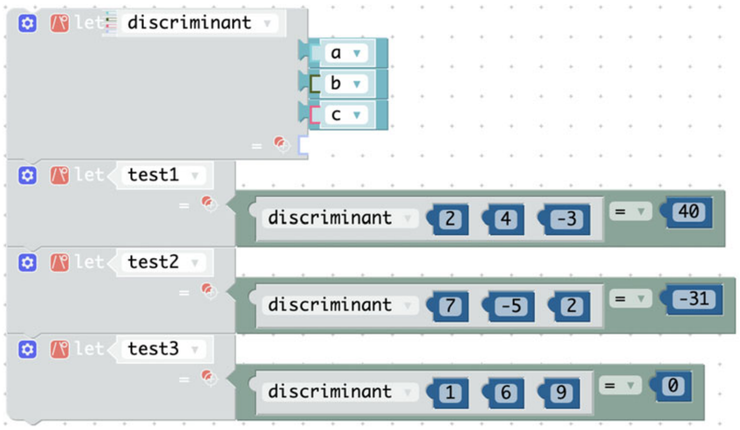


Fig. 8. Header and tests for a function `discriminant`. The type of `discriminant` is constrained by the test cases.

and wished to change the body of `f` to `2 * (x + a)`. To insert blocks for `2 *`, the user first has to move the blocks for `x + a` somewhere else. However, since `x + a` contains free variables, they cannot be placed in the main playground. Instead, the user can keep them in the scope playground, construct blocks for `2 *`, and then restore the blocks for `x + a` afterwards.

3.2 Type checking

In OCaml Blockly, type checking is performed for every user action, avoiding any ill-typed program to be constructed. Furthermore, the type information is shown as the shape of connectors.

Some students remember shapes of base types gradually and use them to avoid plugging a block having a different type. Even though it is unrealistic to parse the complex shape of a connector for a structured type, students recognize different shapes and can identify type mismatches quickly. In particular, the shapes are effective in helping students guess the order of arguments of higher-order functions such as `List.map`.

Since OCaml Blockly knows the field types of declared records, changing the definition of a record affects all the uses of that record. If one adds a new field in a record declaration, all the corresponding record blocks are extended with the new field. This is particularly useful when a record plays a central role in a program, such as world-passing programs for interactive games (Felleisen *et al.*, 2014), where the state of a game is typically represented as a record (called a world). When one extends a world record with a new field, one would naturally be urged to fill in the new fields for all the records that follow.

In the presence of on-the-fly type checking during editing, the program development based on the design recipe (Felleisen *et al.*, 2001) leads to an interesting user interaction. The design recipe advocates turning worked examples into tests before constructing the function definition. As an example, consider the program in Figure 8. The program has a header of the function `discriminant` that computes $b^2 - 4ac$ given three (integer) coefficients a, b, and c, as well as three test cases of that function. Since the student writes tests

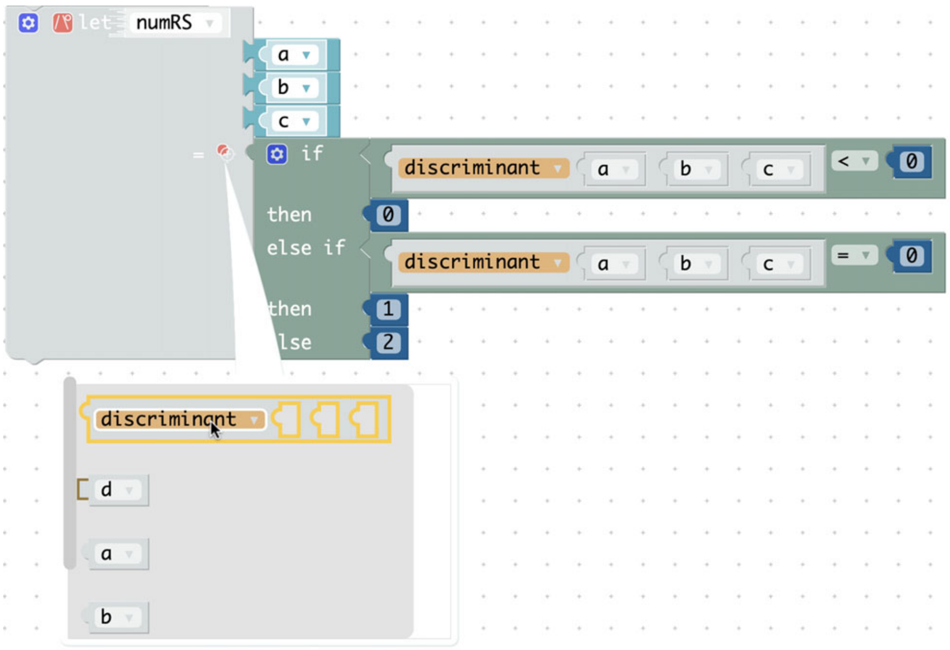


Fig. 9. A function `numRS` that returns the number of real solutions, given coefficients *a*, *b*, and *c*. Students never forget to pass arguments to `discriminant`.

first, the type of the function is constrained. Although the connectors of parameters and the function body have square-shaped polymorphic types because of the let-polymorphic type inference, if one tries to connect a string block to the body of the function, it is rejected because the test cases require the return type to be an integer. By writing tests first, students obtain additional feedback when writing a body of a function.

3.3 Functions

In OCaml Blockly, dragging the name of a function definition creates a function call block having one hole per argument. Having holes for arguments by default makes an obvious thing clear that is nonetheless important for beginning students: one needs to fill in arguments to call a function. Beginning students often forget to pass arguments to a function, especially when a function is called in another function. Figure 9 shows such an example: here, a function `numRS` returns the number of real solutions for a quadratic equation $ax^2 + bx + c = 0$ given the coefficients a, b, and c. To define the function, students use the `discriminant` function they have just defined. With a text editor, many students write the following incorrect program, forgetting the arguments to `discriminant`:

```
(* given a, b, and c, returns number of real solutions *)
(* numRS : int -> int -> int -> int *)
let numRS a b c =
  if discriminant < 0 then 0
  else if discriminant = 0 then 1
  else 2
```

For this program, the OCaml interpreter reports a type error saying that the first `0` should have type `int -> int -> int -> int`, which is indecipherable for beginning students. With OCaml Blockly, students never forget to pass arguments to `discriminant`, because the block for calling `discriminant` has three holes waiting to be filled in.

3.4 Limitations

OCaml Blockly is designed to support static aspects of programming before execution. It has various limitations, too.

Lack of type annotations. OCaml Blockly supports let-polymorphic type inference, but it does not support type annotations. The design recipe advocates that students write the type of a function first. The lack of type annotations (or a way to show types of variables and functions) could have negative effects on students' learning, such as failing to write test cases that are consistent with the type of the function. Currently, one has to open a comment box and write type information there as a comment. The type information from the design recipe page (Section 4.2) is imported as a comment, but it is not shown unless the comment box is opened by the user.

Enforcement of scoping rules. The design to enforce the scoping rule of OCaml all the time has some usability issues. One cannot simply remove a variable definition if the variable is used in the following blocks, since removing the definition leads to an unbound use of the variable. In the current implementation, a variable definition can be removed if it is not used in the following blocks. If it is used, the definition as well as all the following blocks are deleted. We could have implemented the deletion in such a way that all the uses of a variable are deleted when its defining block is deleted, but we have not been able to do so.

Similarly, it is hard to reorder definitions. Ideally, we would want to have the dependency graph and manipulate it arbitrarily as long as the dependency is not violated. However, it would not be easy to build such a tool nor is it clear if such a tool is actually usable for beginning students.

The same connector shape for records. The type of a block is reflected to its connector shape. However, a single shape is assigned for all the record types because reflecting the types of all the fields of a record appears to be unrealistic. Colors could be introduced to distinguish different record types (as for polymorphic types), but it is not implemented yet. A related problem is the use of a general square-shaped connector (with a color) for a polymorphic type. Since a connector for a polymorphic type accepts a block of any type, some students wrongly think that blocks with any differently shaped connectors can be sometimes connected.

Syntax errors. Although OCaml Blockly encourages students to avoid syntax errors, the current implementation of OCaml Blockly also allows students to run incomplete programs, that is, programs with holes. When they do, OCaml Blockly signals an error concerning problems with the program's syntax. OCaml Blockly does not currently offer a way to detect if there are any open connectors that are not filled yet or to jump to the next unfilled connector. It should not be difficult to implement such support, but we have not tried it yet.

Type errors. Since OCaml Blockly does not allow construction of ill-typed programs, it rejects connection of blocks that leads to a type error. When a user tries to connect a block with a conflicting type, a type error message is shown in a pop-up message similar to the one shown by the standard OCaml compiler, which is not user friendly. It would be an interesting direction for future work to see if the OCaml type debugger (see Section 4.2) can be incorporated into OCaml Blockly, so that whenever a user tries to connect a block with a conflicting type, the user can navigate through the program to see why the type mismatch occurred.

4 A functional programming course

In this section, we present the background on the functional programming course and describe how OCaml Blockly was introduced to the course.

4.1 Course background

The functional programming course at Ochanomizu University was launched in 2002 and has been given every year ever since (except for 2004 when the author was on sabbatical). The purpose of the course is to learn how to write correct programs based on the design recipe (Felleisen *et al*., 2001), to experience how the use of various data structures impact computational complexity of programs, and to write somewhat large and useful programs.

The course is targeted at second-year undergraduate students[2] majoring in computer science. All students enter the course with experience using the C language, and most of them have no prior experience with functional languages. The course lasts 15 weeks, and in each week, there is a 90-minute lab-style class held in a computer room.

Table 1 shows the topics covered in the course, as well as students' perspective of the course. In the first lecture, the author tells the students that they will eventually implement the shortest path problem for the Tokyo[3] metro network. Showing such a goal (which might sound a bit ambitious to most students at this stage) gives them a stronger motivation to complete the course. It also helps them understand for what purpose each topic is taught.

Although the course uses OCaml as the implementation language, learning the OCaml language itself is not the goal of the course. Rather, it emphasizes how to systematically construct a program following the design recipe, regardless of the programming language used. As such, a design recipe is introduced at an early stage (week 2) and is used throughout the course.

[2] About 40 each year, all female because Ochanomizu University is a women's university.
[3] The course started with the shortest path problem for the Tokyo metro network, but over the years, more and more station data were added to finally cover almost all the train stations in Japan now.

Table 1. Syllabus for the functional programming course in Ochanomizu University. The rightmost column shows students' perception towards writing the shortest path problem for the Tokyo metro network

Week	Topics covered	Students' perception toward the shortest path problem
1	Variable/function def.	(introduction)
2	If, design recipe	(design recipe)
3	Record, template	Represent station (graph) information as a record
4	List, recursion	Use linear search to obtain the distance between two adjacent stations
5	More recursion	Remove duplicate stations that belong to more than one line using modified insertion sort
6	Dijkstra's algorithm	Construct an initial graph for Dijkstra's algorithm from a list of station names
7	Map	Implement the relax function and apply it to all the tense edges
8	Filter, fold	Implement a function that extracts the station with the minimum distance
9	General recursion	Complete the first version of the shortest path problem (but it is too slow for large data)
10	Tree	(toward using tree search)
11	Exception	Raise exception when the input station is not found
12	Module	Replace linear search with tree search to speed up the program implemented in week 9
13	Sequential execution	Output the shortest path in a pretty way
14	Mutation	(memoization)
15	Heap	Incorporate heap to further speed up the program

4.2 How OCaml Blockly was introduced

Since its initial inception in 2002, we have invested considerable amount of efforts into the course.

Textbook, video, and flipped class. To acquire real knowledge, students need not only to write programs but also to read a textbook to consolidate their understanding. For this purpose, the present author wrote a textbook (Asai, 2007). It is intended to be a Japanese translation of (the first edition of) *How to Design Programs* (Felleisen *et al.*, 2001) although the contents are completely different except for the design recipe.

On top of the textbook, the author prepared a short video[4] around 20 minutes for each week, and turned the class into a flipped class in 2014 where students are required to watch the video, read a designated part of the textbook, and answer a few simple questions before the class begins. During the class, students work on programming assignments with the assistance of the author and teaching assistants.

The textbook and the video were prepared long before OCaml Blockly was developed. Thus, all the code in the textbook and the video are in the text form, not blocks. The author did not update the textbook and the video to use a block-based environment, because he

[4] Available in Japanese at `http://pllab.is.ocha.ac.jp/~asai/book-mov/`.

regarded OCaml Blockly as an optional assistance tool to the course and ultimately, he wanted students to be able to write programs using a text editor.

Design recipe page. To encourage students to follow the design recipe, the author created a web site[5] in 2012. Although it is written in Japanese, the reader would see how the program template at the bottom is automatically created according to the function name, purpose statement, parameter names and their types, and tests that are all input by students. Students can then copy the resulting program to the input area of OCaml Blockly and convert the program into blocks. Despite being extremely simple, the design recipe page enormously improved readability of programs submitted by students. A similar system to help students follow the design recipe is created for Scala by Nose *et al.* (2022).

The design recipe page is a simplistic and minimal approach to enforce the design recipe methodology. Various work has been done to further enhance the methodology and see its merits. To encourage students to write good tests first, Politz *et al.* (2014) develop CaptainTeach, a tool that forces students to first submit a test suite and then to review other students' test suites, before submitting an implementation (which is also peer-reviewed by other students). Wrenn & Krishnamurthi (2020) observe that students voluntarily write tests that are correct and catch as many subtle bugs as possible when given a proper infrastructure. Ren *et al.* (2019) analyze TA office hours and show that the design recipe provides a useful initial view on how students use TA hours.

Type debugger and stepper. To help students debug type errors, the author developed a type debugger for OCaml, in collaboration with Kanae Tsushima (Tsushima & Asai, 2013), Yuki Ishii (Ishii & Asai, 2014), and Naho Wakikawa (Wakikawa & Tsushima, 2018). As with Chitil (2001), the type debugger navigates students to the true source of type errors through a series of questions. If students correctly answer the questions (which is a big "if" though), it always identifies the source of the type error.

To help understand runtime behavior of programs, the author implemented an algebraic stepper (Clements *et al.*, 2001) for OCaml together with Youyou Cong (Cong & Asai, 2016), Tsukino Furukawa (Furukawa *et al.*, 2019), and Hinano Akiyama (Asai & Akiyama, 2025). The stepper enables students to observe how their programs are executed (including non-termination) without launching a sophisticated debugger. We have integrated the stepper into OCaml Blockly, so that students can step-execute their programs directly from OCaml Blockly (shown in textual syntax, not in block syntax).

OCaml Blockly. We have seen three tools we developed in the past: the design recipe page supports constructing a skeleton of a program, and the type debugger and stepper cover the static and dynamic semantics of programs. With all these infrastructures, the author expected students could be able to manage errors by themselves and concentrate on the programming itself. They could not. Where do they stumble? Syntax. If students could not construct a complete program that compiles, the tools introduced into the course are useless. To this end, the author introduced OCaml Blockly originally developed by Haruka Matsumoto (Matsumoto & Asai, 2019).

[5] Available in Japanese at `http://pllab.is.ocha.ac.jp/~asai/book-mov/recipe.html`.

Table 2. Number of students who enrolled, got credit of the course

	2019	2020	2021	2022	2023	total
Enrollments	41	47	47	44	45	225
Got credit	35	39	41	34	44	193

5 OCaml Blockly in the classroom

OCaml Blockly has been used in the functional programming course since 2019. By introducing OCaml Blockly, we did not mean to abandon text-based editing. In a CS-major department, students should become able to write programs in a text editor eventually. However, they do not necessarily have to do so from the beginning, when they are not familiar with the language. In the functional programming course, we first offer OCaml Blockly to get them acquainted with OCaml. Once they understand the syntactic structure of OCaml in the class, they would be able to write programs in a text editor more easily than trying it from day one. We intend that OCaml Blockly serves as a friendly introduction to text-based editing as well as typed and functional programming. In this section, we investigate how OCaml Blockly was used by students and how it helped them.

At the end of each course, we asked all the students to answer questionnaire on OCaml Blockly (as well as other tools and the course in general). The questionnaire was not anonymous, but it was explicitly stated that the answer did not affect the final grade and students got a small bonus point by answering it. In this section, we analyze the questionnaire and try to answer the following research questions:

RQ1 When did students stop using OCaml Blockly?
RQ2 Did students find OCaml Blockly useful?
RQ3 Did programming become easier using OCaml Blockly than using a text editor?
RQ4 Did OCaml Blockly have negative effects on text-based programming?

Table 2 shows the number of enrolled students and the number of students who actually earned the credit. These numbers are the same for other tables that follow. The courses for 2019, 2020, and 2023 were in-person while the ones for 2021 and 2022 were online only due to COVID-19 pandemic. In addition, in 2023, we interviewed selected students to hear how they perceived OCaml Blockly in detail.

5.1 When students stop using OCaml Blockly

At the beginning of each iteration of the course, we announced that students were free to use either OCaml Blockly or a text editor (Emacs equipped with the type debugger and the stepper) and that they would have to switch to a text editor at week 10 because OCaml Blockly does not support user-defined data types (trees). Table 3 shows when students stopped using OCaml Blockly. The choices are at an early stage (around week 1), in the middle of the course when programs became gradually larger (around week 6), when students wrote the first version of the shortest path problem (around week 8), when trees (which are not supported in OCaml Blockly) were introduced (around week 10), and even

Table 3. Questionnaire 1: when did you stop using OCaml Blockly?

	2019	2020	2021	2022	2023	total
Around week 1	8	18	9	4	3	42
Around week 6	7	10	8	3	6	34
Around week 8	6	5	7	7	6	31
Around week 10	7	7	9	15	8	46
After week 10	0	0	1	3	0	4
Other	1					1

after week 10. (The response for the "other" row said that the student did not use OCaml Blockly at first but started to use it in the middle of the course only to find a text editor was easier to use when she already got used to the syntax of OCaml.)

The table shows that students' responses vary. Some students did not use OCaml Blockly at all or abandoned it at an early stage. According to the free-text responses to the questionnaire, they were already familiar with text-based editing (which they did in the first year programming courses) and thought that they could write programs in a text editor. One of them stated that once she understood syntax, text-based editing was faster. Another reason some students did not use OCaml Blockly is that the textbook and the video assumed a text-based environment. One student wrote (and stated in the interview) that she wanted to use the same environment as in the textbook and the video. She also said in the interview that although instructor's explanation using OCaml Blockly was easier to understand than using text-based environment, she wanted to compare the resulting textual code with what she wrote by herself.

Even though all the students had experienced text-based editing already, most of them gave OCaml Blockly a try. Among them, roughly one-third of them stopped using OCaml Blockly as programs became larger. They wrote that at that point, they began to understand the syntactic structure of OCaml programs, and as a result, text-based editing was faster. On the other hand, one student wrote that she switched to text-based editing at this stage because she found she had trouble writing the same program in a text editor and thought she needed training on using a text editor.

Rather surprisingly, a certain number of students used OCaml Blockly all the way to complete the first version of the shortest path problem. Since they did not report how much effort it took to write such a large program in OCaml Blockly, we asked them during the interviews. One of the students responded that the size of a program did not matter; it was hard to read a program when its block representation became large, but it would be the same for textual programs since they also became long; then, the structure of programs was clearer with a block interface. Another student said she did not remember the precise syntax—because she did not have to—and thus continued using OCaml Blockly as long as she could. Yet another student responded that she could understand code in blocks visually, while she was confused when looking at textual code. Students who came that far appeared to acquire their own way of understanding programs in blocks regardless of the size.

Even after week 10, some students used OCaml Blockly. From the comments in the questionnaire, they used it to see how complex constructs, such as nested local variable definitions, were written in OCaml. In the interview, one student said she used OCaml

Table 4. Questionnaire 2: was block interface useful?

	2019	2020	2021	2022	2023	total
Did not use it	3	5	3	4	2	17
Not useful at all	1	0	1	0	0	2
Not very useful	2	4	1	0	1	8
Sometimes useful	4	16	6	7	7	40
Very useful	11	11	20	20	16	78

Blockly to construct parts of a program that were still supported by OCaml Blockly and wrote remaining parts of the program in a text editor, thus confirming the supported parts were type checked.

5.2 Whether students found OCaml Blockly useful

Table 4 shows whether students thought the block interface was useful in general, compared to text-based editing. As with the previous question, some students responded they did not use OCaml Blockly at all. Among those who used OCaml Blockly, some students found OCaml Blockly not (very) useful, saying that it was good only for very small programs. Still, one student in this group said it was easier to recognize the structure of programs with OCaml Blockly.

A majority of students who used OCaml Blockly found it was useful. The free-text responses typically mentioned the low barrier for beginning students, such as:[6] "It was easy to use," "At the beginning, it was not realistic for me to write programs in a text editor," "I could write programs without remembering syntax," "I could see/understand the structure of programs," and "OCaml Blockly supported understanding of basics (for those who are not good at programming)." There were deeper comments, too. One student wrote that she could concentrate on programming without being bothered by syntax. Another student wrote that once she was used to OCaml Blockly, she could write programs in a text editor more easily.

Rather unexpectedly, a fair number of students said OCaml Blockly was useful for detecting type errors, in addition to syntax errors. Typical responses said that they could notice type mismatches intuitively when a block did not connect. Another use of the type information was to guess the ordering of arguments of higher-order functions, such as `List.map`, according to the shape of the arguments' connectors. One student went even further and wrote that she could be more aware of types when programming in a text editor after using OCaml Blockly.

To further assess the effect of connector shapes, we tested whether students remembered the shape of connectors in the interviews. We showed to students connectors for integers, floats, strings, booleans, pairs, lists, functions, and records and asked them what these connectors were for. From the interviews, it turned out that (1) a student who abandoned OCaml Blockly early (naturally) did not remember connector shapes very well, (2) for students who used OCaml Blockly until later, the results were diverse: some of them

[6] These responses are author's translations of the original responses in Japanese.

Table 5. Questionnaire 3: do you feel the programming became easier (e.g., for understanding syntactic structure, typing) by using block interface?

	2019	2020	2021	2022	2023	total
Strongly no	2	0	1	0	0	3
No	0	1	2	1	0	4
Neutral	5	12	3	4	6	30
Yes	6	18	14	7	7	52
Strongly yes	8	5	11	16	13	53

remembered very little, some of them remembered the shapes for base types, and some of them could answer most of the connector shapes. One student who remembered most of the connector shapes said she might have recognized simple shapes during programming, but she did not see the structured types carefully. Rather, she tried to connect a block to see if type matched. One factor that affected this behavior could be polymorphic types. Since the shape for polymorphic types is the same (although they have different colors), she said she tried to connect a block even if connector shapes were different.

These responses suggest that the shapes for simple types are somewhat useful while shapes for complex types are not very important. Even if shapes do not match, students often try to connect a block and discover type mismatch when the program is rejected. This behavior could lead to a trial-and-error programming where students try to connect a block until it fits without thinking about type mismatch very much. In fact, one student wrote in the free-text responses that she could often go through without thinking deeply. Although we think it should not be a serious problem (because the student also wrote that the type mismatch did trigger thinking about types), we think it is important to encourage students to reflect on what they are doing.

One unexpected finding from the interviews was that almost all of them recognized types of blocks by their colors. Students unanimously mentioned colors, one of them even saying she felt the 'atmosphere' of blocks from their colors. Although colors in OCaml Blockly are assigned mostly based on types, it is not strictly so. For example, when an integer block is connected to a `then` branch of an `if` block, the connectors of both the `then` and `else` branches change their shapes to the one for integers, but the color remains the same as before. Similarly, the `int_of_float` block has the same color as an integer block (because it returns an integer), even though it receives a float argument. This observation suggests it might be effective to assign not only the same shape but also the same color to both the male connector and female connector of the same type.

5.3 Whether programming became easier in OCaml Blockly than in a text editor

Table 5 shows whether students thought programming became easier by using OCaml Blockly than using a text editor. There were some "strongly no" and "no" responses. Most of them were from those who did not use OCaml Blockly at all. One student wrote that the programming did not become easier because she wrote programs in a text editor anyway even when she used OCaml Blockly, thus doubling the time for programming. (Apparently, she did not copy textual programs OCaml Blockly produced.) Another student wrote that she understood what she was doing only when she wrote programs in a text editor.

30 students (21%) among 142 students who participated in the questionnaire stayed neutral to this question. Most of them did not provide free-text comments, but submitted comments showed that they found OCaml Blockly useful but were not sure if it led to (easier programming with) deeper understanding. One student wrote OCaml Blockly may have enabled her to understand the syntactic structure and typing of programs. Another student wrote she sometimes stopped thinking because OCaml Blockly was too useful and she could write programs without much thought.

Most other students found OCaml Blockly made programming easier for understanding syntactic structure and typing. The free-text comments can be classified into two categories: rule enforcement and empowerment. The first category is further divided into two groups: syntactic structure and typing.

Students appreciated the syntactic support for complex constructs, such as (nested) conditionals, pattern matching, records, and local variable definitions. They also mentioned that they never passed wrong number of arguments to a function. The syntactic structure of programs becomes obvious once students get used to the language. Nonetheless, the responses indicate that it is important especially in the early stages of programming. For instance, at first, they do not know how to write a simple conditional, where it can appear, and how it connects to other constructs.

The vast majority of free-text comments were about typing. OCaml Blockly does not allow composition of blocks having incompatible types. Whenever a block is rejected for connection, students are forced to think why it is rejected. Students wrote: "I learned all branches of conditionals/pattern matching must have the same type," "I could avoid making type errors," "I could see/understand types visually in OCaml Blockly," and "I became used to think about types in general." The original motivation for designing OCaml Blockly was to support syntax, but it turned out pleasantly that the support for typing (and scoping) was also quite effective.

The second category of students' responses is empowerment. Beginning students who are not confident in their ability to program are particularly empowered and encouraged to go forward. Students wrote: "I thought programming was difficult for me, but I could think about it positively," "The way only the syntactically legal and well-typed blocks could be connected made me learn especially in early stages how to write a program," "I could avoid thinking about inessential details such as matching parentheses," and "I became able to structure a program in my mind more easily."

5.4 Whether OCaml Blockly had negative effects on text-based programming

Table 6 shows whether students thought they were spoiled by the use of OCaml Blockly. Around 72% of students thought OCaml Blockly did not have negative effects ("strongly no" or "no"). Some of them did feel confused when they switched to text-based programming, but they became used to it very quickly. Overall, the benefits of having easier introduction and syntax/typing support exceeded a small struggle at transition time.

There were many positive and also fundamental comments: "I could get used to OCaml quickly via the block interface," "Because of OCaml Blockly, I could catch up with the

Table 6. Questionnaire 4: do you feel your programming ability decreased (e.g., cannot live without block interface, cannot program in a text editor) because of the block interface?

	2019	2020	2021	2022	2023	total
Strongly yes	0	0	0	0	1	1
Yes	0	0	1	3	1	5
Neutral	5	10	6	7	6	34
No	6	10	10	11	7	44
Strongly no	10	16	14	9	11	60

class," "There were more benefits than drawbacks," "By using OCaml Blockly, I learned how to write programs in a text editor," "Since it was the structure of programs that was important, the interface did not matter," "The flow of thought is the same in both the interfaces," and "I became able to program in other programming languages, too."

On the other hand, there were a few "strongly yes" and "yes" responses as well as a certain number of "neutral" responses. The students who responded "yes" wrote they did not remember how to write programs (even if they remembered typing rules), especially for complex constructs such as pattern matching and records. The student who responded "strongly yes" did not leave any written comments, but in the interview, she made the same point: because she did not write textual programs with the block interface, she simply could not write any programs in a text editor and needed some practice when she switched the interface. On the contrary, students who responded "(strongly) no" typically wrote they learned concrete syntax from the textual programs produced by OCaml Blockly. Students did have to handle programs in text files to send them to the check system provided by the author or to submit them as part of assignments. During this process, they had the opportunities to see text-based programs. Still, the above responses indicate that it would be important to urge students to look into text-based programs from time to time. In fact, one of the students said in the interview that she found it important to be able to see textual programs in the OCaml Blockly window.

The "neutral" responses expressed concerns on the possibility of using OCaml Blockly without deep understanding. That is, OCaml Blockly could be too useful for them so that they could rely on it too much. Although many students among them wrote they could transition to text-based programming rather smoothly possibly with some struggle, it would be important to remind them to reflect on what they have been doing in each assignment.

Transition to text-based programming. Since the questionnaire did not contain a question on the transition to text-based programming explicitly, we asked in the interview how students felt about the transition. Almost all of the interviewees said OCaml Blockly was nice at the beginning. In other courses, they typically learn syntax first. With OCaml Blockly, they said they could get over it without any difficulty. One student stated it would have been difficult to write a program for the shortest path problem if text-based programming was used from the beginning.

On the other hand, one student said she was confused by the concrete text-based syntax of various constructs, such as conditionals, because she used conditionals of other programming languages in other courses. Another student responded it could be better if the transition was scheduled a bit earlier. She also said it would have been nice if she could use blocks and practice writing programs without blocks at the same time. Despite these responses, they stated that they became able to write textual programs at the end and that they thought their ability to write textual programs would not be lower than the case they had started programming in a text editor from the beginning.

We also asked how they felt about restricting what they can write in OCaml Blockly, i.e., they could write only legal programs as opposed to text-based programming where they could write any programs freely. Most of the interviewees responded they had no negative opinion about it. Rather they found it positively, saying they could understand what is wrong with their program when it is rejected. Other student responded she did not feel any inconveniences because she got use to think about programs under the restrictions of OCaml Blockly. Yet other student stated text-based programming was too free; if there were rules, it was better to see the error beforehand rather than at compile time. On the other hand, one student answered she preferred to write programs as she wanted and receive errors after she completed her programs; when she could not connect a block to some place, she was stuck at that point and could not go forward.

To further see the effect of OCaml Blockly on text-based programming, Mai Kitagawa (Kitagawa & Asai, 2021) conducted a preliminary analysis on two aspects of the use of OCaml Blockly in the functional programming course: how many errors students produced and how much time they spent on assignments. First, we compared the number of errors (including syntax errors, unbound variables, and type errors) produced after week 10 during the 2019 course against the average number of errors produced after week 10 during the courses conducted in 2015 to 2018. The latter courses were offered without using OCaml Blockly. For weeks other than 11, the number of errors for 2019 was smaller than the previous years. For week 11, the number of errors for 2019 was slightly greater. Although no statistically meaningful results can be drawn here, at least introduction of OCaml Blockly for the first 10 weeks did not appear to cause students to face more errors when switching to text-based programming.

Second, we compared the median time to complete assignments after week 10 (the time between a function name for an assignment first appeared in a submitted program and when it passed the check system). Again, there were no notable differences, with year 2019 being mostly faster than the average of years 2015 through 2018 and slightly slower for some cases. With these observations, we consider introduction of OCaml Blockly does not hamper learning of programming, including text-based programming.

5.5 Threats to validity and limitations

To begin with, the sample size is small, around 40 students per year over five years. Thus, the results in the paper might reflect these particular students rather than general students. The fact that all students are female could also have affected the results. Furthermore, the students knew that OCaml Blockly was developed by the author's research group, hence they might have answered the questions in a more positive way than necessary.

Although we would ultimately want to see if the use of OCaml Blockly has any positive impacts on students' programming ability, what the questionnaire asked is students' perception of OCaml Blockly. One way to assess students' programming ability is to see whether students perform better in the follow-up courses. We have not done this for two reasons. First, we think it is not ethical to force the use of either OCaml Blockly or a text editor to students. Second, the follow-up course (a programming language course on either an interpreter or a compiler depending on the year of the course) is one year away, and the number of enrollments of the course is very small (typically three to five) to do comparison experiments.

6 Related work

Many block-based programming environments have been developed such as Alice (Pierce *et al.*, 1998), Scratch (Resnick *et al.*, 2009), Blockly (Fraser, 2015), and Snap! (Harvey & Mönig, 2010). They are widely used throughout the world not only for introductory programming but also for game programming, simulation, and many other topics. As block-based programming environments, they all provide programming experiences without syntax errors. On top of it, OCaml Blockly offers an environment without scoping and typing errors in a beginner-friendly way.

6.1 Blockly and related environments

The original Blockly (Fraser, 2015) developed by Google is an extensible framework for realizing block-based environments for various imperative languages. It supports all the basic features required for block-based environments, including statement and expression blocks, dialog buttons, and conversion to text programs. OCaml Blockly was forked from Google Blockly in April 2018 and inherited all the features of Google Blockly. We added OCaml-specific features, in particular, the scope playground and let-polymorphic type inference. Since the added features required substantial changes to Google Blockly, however, it became unrealistic to merge the upstream changes into OCaml Blockly. As such, OCaml Blockly is now an isolated branch missing features recently added to Google Blockly, such as shadowed blocks.

HenBlocks (Boey & Adams, 2022) is a Blockly-based structured editor for the Coq proof assistant. It provides syntactic support for writing proof scripts in Coq, including inductive datatype definitions, function definitions, theorems, and proofs. Being block based, it naturally prevents syntax errors. It also supports α-renaming of variables and names of assumptions, respecting the scope rules of Coq. Since Coq supports function definitions, one can write functional programs in Coq. However, type checking is deferred to proof checking time. Thus, one can construct syntactically correct but ill-typed programs that result in a type error at proof checking time.

BlockPy (Bart *et al.*, 2020) is a block-based programming and execution environment for Python. It features a dual text/block interface, and it has been shown that the block interface is helpful for novice learners in a computational thinking class for non-CS major students. Being a block-based environment for Python, it does not have functionalities equivalent to OCaml Blockly's scope playground or type checking. BlockPy uses a block language that is suitable for novice learners (e.g., `for each`) instead of strict Python syntax (e.g., `for`), while OCaml Blockly uses original keywords so that program texts on blocks are identical to the textual programs (except for the automatic insertion of parentheses).

6.2 Connectors as representation of types

The idea of representing precise type information as shapes of connectors is considered in TypeBlocks (Vasek, 2012). In TypeBlocks, a connector shape of a composite type consists of the shapes of the constituent types. OCaml Blockly, developed independently of TypeBlocks, employs the same approach for pair, list, and function types, but not for record types. On one hand, precisely representing types of fields of a record requires a large connector. Even a connector for a simple pair of integers requires the height more than twice the size of an integer connector. On the other hand, using a single shape for all the records prevents us from distinguishing different record types by looking at the connector shape. The design choice to use a fixed connector for records and complex connectors for other data in OCaml Blockly came from the ease of implementation. It is yet to be seen whether students benefit from more information on the record connectors.

Another approach to showing type information is proposed by Poole (2019). He presents a block design for Haskell in which type information including type classes is shown to users. He uses labels and colors for types and black hatch patterns for polymorphic types. Types for lists, tuples, and functions are represented as a white base on which component types are shown. Because labels are used instead of connector shapes, the type information does not take too much vertical space for complex nested types.

6.3 Integrated development environment

There are a number of integrated development environments (IDEs) for various languages. They allow various degrees of spell checking, selection of bound variables and library functions, showing type information and documentation, and even suggesting interpolation of programs. Notably, Visual Studio Code (VS Code)[7] supports many programming languages including OCaml. The language server for OCaml used in VS Code (and also other text-based editors, such as Emacs and Vim) is Merlin (Bour *et al.*, 2018). It provides services such as signaling syntax errors, typing errors and warnings, or making auto-completion suggestions as well as more advanced features such as asking the type of a variable under the cursor or going to the definition. Merlin provides them in response to every user action by implementing incremental lexing, parsing, and type checking for an incomplete program.

VS Code with Merlin would be ideal once one gets used to the basic structure of OCaml programs. However, VS Code is not necessarily friendly to beginners and those who are not confident in their programming ability. OCaml Blockly offers an introduction to programming for those beginning students by showing what constructs are available in a menu, how complex constructs are built, how the use of a variable is restricted within its scope, and how type checking is performed.

[7] `https://code.visualstudio.com`.

6.4 Structured editors

The idea of a structured editor was first proposed by the Cornell Program Synthesizer (Teitelbaum & Reps, 1981). It was intended not necessarily for beginners but also for experts.

More recently, Omar *et al.* (2017) designed Hazel, a web-based live programming environment for an ML-like functional programming language. It is equipped with a structural editor that understands the syntax and typing rules of the language. One of the unique features of Hazel is it assigns abstract syntax trees (ASTs) to any programs including incomplete ones (and even allows us to run them). Hazel does not exclude syntactically incorrect or ill-typed programs. The internal ASTs are built according to the textual representation of programs. In contrast, OCaml Blockly does not allow syntactically incorrect or ill-typed programs. Textual representation of programs is derived only when there are corresponding internal ASTs (possibly with holes).

Whether the main representation is textual or structural affects user experience. Consider a function `add1` that adds 1 to its argument.

```
let add1 x = let y = 1 in x + y
```

In Hazel, renaming the first `y` to `x` results in

```
let add1 x = let x = 1 in x + y
```

where `x` in `x + y` is rebound to the new let-bound `x`, and `y` becomes unbound. This is because the AST is built according to the textual representation of the program. In OCaml Blockly, renaming of `y` to `x` is prohibited, because it shadows `x` that is used in `x + y`. (If `x` is not used in the body of the let expression, the renaming is allowed because it does not shadow any binding.) In OCaml Blockly, the binding is at the AST level, and it cannot be broken by renaming. If one wants to change the binding, one has to delete the variable and create a new variable by dragging from its declaration.

Racket (Felleisen *et al.*, 2018) has the integrated development environment DrRacket (Findler *et al.*, 2002) that is text based but has a "Check Syntax" button (and a stepper among others). After checking syntax and scoping, DrRacket analyzes declarations and uses of variables and shows arrows between them when a user hovers a mouse on a variable. By properly setting `#lang` directive, the checking of syntax can be done continuously in the background. It also allows renaming of bound variables. If we try to replace `y` to `x` in the `add1` function (rewritten in Racket), DrRacket warns that the renaming conflicts with an existing variable name.

The enforcement of all the static rules (scoping rules in particular) employed by OCaml Blockly inhibits users from constructing ill-formed programs temporarily. However, to edit a well-formed program into another well-formed program, one often needs to go through ill-formed programs. Such editing is not permitted in OCaml Blockly. For more flexible uses of variables without violating scoping rules, OCaml Blockly supports the scope playground. There appears to be no similar mechanisms in block-only environments, but there are some work on allowing intermediate ill-formed programs in hybrid environments and structured (projectional) editors.

Frame-based editing (Kölling *et al.*, 2015) takes a hybrid approach. Statement blocks (called frames) are structured enforcing scoping rules, while expressions are input as texts, allowing users to write temporally ill-formed expressions. MPS (Voelter *et al.*, 2014) is a projectional editor for expert programmers. MPS operates on program texts which are actually projections of the internal ASTs and users can directly manipulate the ASTs by editing programs represented as texts. Gradual structure editing (Moon *et al.*, 2023) follows MPS in that a program text is always a projection of the internal AST. It avoids unexpected editing behavior of MPS by introducing obligations that have to be satisfied for the currently input program text to be structurally correct.

7 Conclusion

Among many block-based programming environments, OCaml Blockly is unique in that it understands not only the syntax but also scoping and typing rules of the functional language OCaml. It has been used in a functional programming course in Ochanomizu University for several years with various positive feedback.

We have introduced OCaml Blockly to second-year students. In the current form, OCaml Blockly would not be useful for upper-level courses because OCaml Blockly supports only basic constructs and lacks, for example, algebraic data types, exception handling, and modules. Some form of key interface that enables insertion of blocks to designated places would also be preferable, since drag and drop becomes tedious for advanced users. On the other hand, a simple type-based refactoring offered by OCaml Blockly could be useful for advanced users, too, that changes all the instances of a record once the definition of the record is modified.

We have not been able to perform any comparison experiments to assess the effect of OCaml Blockly, as described in Section 5.5. We think it is too much to ask students to use one fixed interface for more than half of the course. If we have chance to offer shorter courses (maybe for non-CS-major students), we could consider comparison experiments to see the effects of OCaml Blockly. However, the benefit of scoping and typing support of OCaml Blockly would be appreciated only after students do certain amount of programming, and we are not sure if short experiments would result in any noticeable difference.

After the graduation of the main developer, OCaml Blockly is maintained almost solely by the present author. As such, it is becoming hard to maintain OCaml Blockly, not to mention, improve it. On the other hand, if we are to have a block interface for typed functional languages, scoping and typing support is beneficial. We hope the work here forms a basis for future development.

OCaml Blockly is developed on GitHub[8] and can be run on a browser.[9]

Acknowledgments

I am grateful to Haruka Matsumoto and all my colleagues who contributed to the development of OCaml Blockly and its implementation in the classroom. Without their

[8] `https://github.com/kenichi-asai/ocaml-blockly/`.
[9] `https://kenichi-asai.github.io/ocaml-blockly/`.

efforts, OCaml Blockly could not have been used in real educational settings. Shriram Krishnamurthi initially suggested writing a paper on OCaml Blockly and provided invaluable advice and encouragement throughout the reviewing process. I would also like to extend my appreciation to the reviewers, whose comprehensive and constructive comments significantly improved the quality of the paper.

Funding

This work was partly supported by JSPS KAKENHI under Grant No. JP20K12107.

Conflicts of interest

The author reports no conflict of interests.

Author ORCID

K. Asai, https://orcid.org/0000-0001-8040-0394

References

Asai, K. (2007) *Foundations of Programming*. SAIENSU-SHA, Tokyo. (In Japanese).

Asai, K. & Akiyama, H. (2025) Algebraic stepper for simple modules. In Proceedings of the 2025 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, pp. 13–29.

Bart, A. C., Tibau, J., Kafura, D., Shaffer, C. A. & Tilevich, E. (2020) Design and evaluation of a block-based environment with a data science context. *IEEE Trans. Emerging Topics Comput.* **8**(1), 182–192.

Boey, B. & Adams, M. D. (2022) Henblocks: Structured editing for coq. In The Coq Workshop 2022, 2 pp.

Bour, F., Refis, T. & Scherer, G. (2018) Merlin: A language server for OCaml (experience report). *Proc. ACM Program. Lang.* **2**(ICFP), 1–15.

Chitil, O. (2001) Compositional explanation of types and algorithmic debugging of type errors. In Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, pp. 193–204.

Clements, J., Flatt, M. & Felleisen, M. (2001) Modeling an algebraic stepper. In Programming Languages and Systems (ESOP 2001), Lecture Notes in Computer Science, vol. 2028. Springer Berlin Heidelberg, pp. 320–334.

Cong, Y. & Asai, K. (2016) Implementing a stepper using delimited continuations. In SCSS 2016. 7th International Symposium on Symbolic Computation in Software Science, pp. 42–54.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2001) *How to Design Programs. An Introduction to Computing and Programming*. The MIT Press.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2014) *How to Design Programs, Second Edition. An Introduction to Programming and Computing*. The MIT Press.

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J. & Tobin-Hochstadt, S. (2018) A programmable programming language. *Commun. ACM* **61**(3), 62–71.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. & Felleisen, M. (2002) DrScheme: A programming environment for Scheme. *J. Funct. Program.* **12**(2), 159–182.

Fraser, N. (2015) Ten things we've learned from Blockly. In 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), pp. 49–50.

Furukawa, T., Cong, Y. & Asai, K. (2019) Stepping OCaml. *Electron. Proc. Theoret. Comput. Sci.* **295**, 17–34.

Harvey, B. & Mönig, J. (2010) Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists? In Proceedings for Constructionism, pp. 1–10.

Ishii, Y. & Asai, K. (2014) Report on a user test and extension of a type debugger for novice programmers. *Electron. Proc. Theoret. Comput. Sci.* **170**, 1–18.

Kitagawa, M. & Asai, K. (2021) Analysis of students learning OCaml. In Proceedings of the 23rd JSSST Workshop on Programming and Programming Languages, pp. 1–16. (In Japanese). Received best presentation award.

Kölling, M., Brown, N. C. C. & Altadmri, A. (2015) Frame-based editing: Easing the transition from blocks to text-based programming. In Proceedings of the Workshop in Primary and Secondary Computing Education, pp. 29–38.

Matsumoto, H. & Asai, K. (2019) Visual programming editor for OCaml based on Blockly. In Proceedings of the 21th JSSST Workshop on Programming and Programming Languages, pp. 1–15. (In Japanese). Received best paper award and best presentation award.

Moon, D., Blinn, A. & Omar, C. (2023) Gradual structure editing with obligations. In 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 71–81.

Nose, J., Cong, Y. & Masuhara, H. (2022) Mio: A block-based environment for program design. In Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E, pp. 62–69.

Omar, C., Voysey, I., Hilton, M., Sunshine, J., Goues, C. L., Aldrich, J. & Hammer, M. A. (2017) Toward semantic foundations for program editors. In 2nd Summit on Advances in Programming Languages (SNAPL 2017). Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 11:1–11:12.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.

Pierce, J., Cobb, T. & Pausch, R. (1998) Alice. In ACM SIGGRAPH 98 Conference Abstracts and Applications, p. 140.

Politz, J. G., Patterson, D., Krishnamurthi, S. & Fisler, K. (2014) CaptainTeach: Multi-stage, in-flow peer review for programming assignments. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE'14), pp. 267–272.

Poole, M. (2019) A block design for introductory functional programming in Haskell. In 2019 IEEE Blocks and Beyond Workshop (Blocks and Beyond), pp. 31–35.

Pottier, F. & Rémy, D. (2004) The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*. The MIT Press, pp. 389–489.

Ren, Y., Krishnamurthi, S. & Fisler, K. (2019) What help do students seek in TA office hours? In Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER'19), pp. 41–49.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. & Kafai, Y. (2009) Scratch: Programming for all. *Commun. ACM* **52**(11), 60–67.

Teitelbaum, T. & Reps, T. (1981) The cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM* **24**(9), 563–573.

Tsushima, K. & Asai, K. (2013) An embedded type debugger. In Implementation and Application of Functional Languages (IFL 2012), Lecture Notes in Computer Science Vol. 8241. Springer Berlin Heidelberg, pp. 190–206. Received Peter Landin Prize.

Vasek, M. (2012) *Representing Expressive Types in Blocks Programming Languages*. Honors Thesis, Wellesley College.

Voelter, M., Siegmund, J., Berger, T. & Kolb, B. (2014) Towards user-friendly projectional editors. In Software Language Engineering (SLE 2014), Lecture Notes in Computer Science, vol. 8706. Springer International Publishing, pp. 41–61.

Vouillon, J. & Balat, V. (2014) From bytecode to JavaScript: the Js_of_ocaml compiler. *Software Pract. Exp.* **44**(8), 951–972.

Wakikawa, N. & Tsushima, K. (2018) Practical type error slicer and its evaluation. In Proceedings of the 20th JSSST Workshop on Programming and Programming Languages, pp. 1–17. (In Japanese).

Wrenn, J. & Krishnamurthi, S. (2020) Will students write tests early without coercion? In Proceedings of the 20th Koli Calling International Conference on Computing Education Research. Article 27, 5 pp.