# Stepping OCaml

Tsukino Furukawa Youyou Cong Kenichi Asai
Ochanomizu University
Tokyo, Japan
{furukawa.tsukino, so.yuyu, asai}@is.ocha.ac.jp

Steppers, which display all the reduction steps of a given program, are a novice-friendly tool for understanding program behavior. Unfortunately, steppers are not as popular as they ought to be; indeed, the tool is only available in the pedagogical languages of the DrRacket programming environment.

We present a stepper for a practical fragment of OCaml. Similarly to the DrRacket stepper, we keep track of evaluation contexts in order to reconstruct the whole program at each reduction step. The difference is that we support effectful constructs, such as exception handling and printing primitives, allowing the stepper to assist a wider range of users. In this paper, we describe the implementation of the stepper, share the feedback from our students, and show an attempt at assessing the educational impact of our stepper.

## 1 Introduction

Programmers spend a considerable amount of time and effort on debugging. In particular, novice programmers may find this process extremely painful, since existing debuggers are usually not friendly enough to beginners. To use a debugger, we have to first learn what kinds of commands are available, and figure out which would be useful for the current purpose. It would be even harder to use the command in a meaningful manner: for instance, to spot the source of an unexpected behavior, we must be able to find the right places to insert breakpoints, which requires some programming experience.

Then, is there any debugging tool that is accessible to first-day programmers? In our view, the algebraic stepper [1] of DrRacket, a pedagogical programming environment for the Racket language, serves as such a tool. The algebraic stepper is literally a stepping evaluator for DrRacket programs. Figure 1 illustrates how the stepper works. In the "before" window (left), we see that the expression `(= 3 0)` is what we are going to reduce at the current step, i.e., the expression is a *redex*. In the "after" window (right), we find the redex is replaced by `false`, which is the result of reducing `(= 3 0)`.

Note that there is a notable difference between the DrRacket stepper and the stepping facility of Eclipse or gdb. While Eclipse's debugger only shows which line is being executed, the stepper tells us how the whole program looks like at each step, by rewriting the input program according to the reduction semantics of Racket. Using the stepper would be easier for beginners, because all they need to understand is how programs are rewritten as the execution proceeds, just like how formulas are rewritten in mathematics. Thus, we see that the DrRacket stepper should serve as an excellent debugging tool for beginning programmers.

Unfortunately, the DrRacket stepper is only available in the "teaching languages"; it does not support the full Racket. What this means is that we cannot step through programs that uses advanced features, including exception handling. This is quite disappointing, since understanding how exceptions are handled is not easy for beginners.

We present a stepper that supports a practical fragment of OCaml, covering most language constructs used in the "Functional Programming" course of Ochanomizu University. The stepper looks exactly like

P. Achten, H. Miller (Eds.):
Trends in Functional Programming in Education 2018 (TFPIE'18)
EPTCS 295, 2019, pp. 17–34, doi:10.4204/EPTCS.295.2

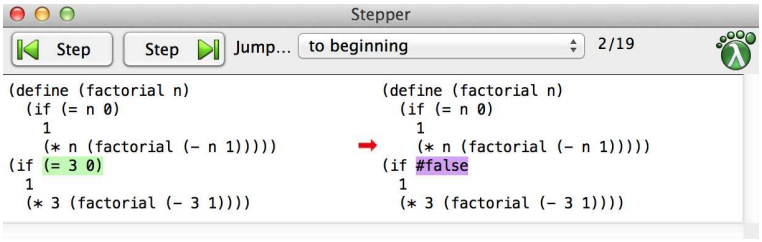


Figure 1: Stepping factorial in DrRacket

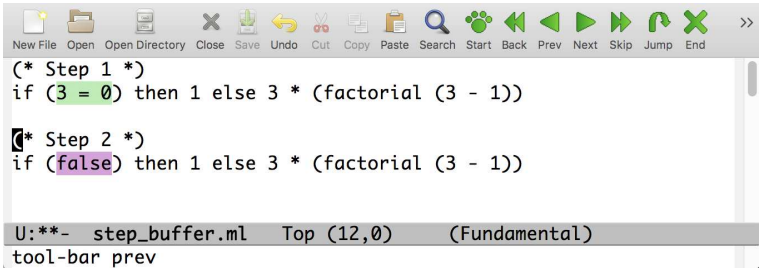


Figure 2: Stepping factorial in OCaml

the one provided in DrRacket, as shown in Figure 2. We build the stepper by making modifications to a standard big-step interpreter. The idea is to keep track of evaluation contexts and to reconstruct the whole program at each reduction step, which are necessary for reconstructing the whole program when reducing redexes. We also share our experience of using the stepper in the classroom, and attempt to evaluate the impact on students' understanding.

The reminder of this paper is organized as follows. In Section 2, we describe the implementation of a mini stepper that supports exception handling. It also shows the overview of the actually implemented stepper. In Section 3, we present how the stepper is used in our classroom, as well as what students found about it in the recent courses. Related work is discussed in Section 4, and the paper concludes in Section 5.

## 2 Implementation

This section presents the implementation of our stepper. We step-evaluate programs in the following way. First, we convert a given program into an abstract syntax tree using the built-in parser of OCaml. Next, we pass the parsed program to a stepping interpreter, which outputs the whole program at each reduction step. The output programs are then processed by an Emacs Lisp program, in such a way that the user can see the steps one by one. Here we focus our attention to the stepping interpreter, which plays the key role in the whole stepping system.

As the stepper is supposed to tell us how a program is evaluated, we have to make sure that it evaluates programs in the same order as the OCaml interpreter does. For the presentation issue, here we restrict ourselves to a toy language, consisting of lambda terms and a simplified version of the try-with construct. The full language supported by the current stepper is presented in Section 2.3.

```
type e_t =
         | Var of string               (* x *)
         | Fun of string * e_t         (* fun x -> e *)
         | App of e_t * e_t            (* e e *)
         | Try of e_t * string * e_t   (* try e with x -> e *)
         | Raise of e_t                (* raise e *)
```

Figure 3: Syntax

## 2.1 Building an Interpreter

In Figures 3 and 4, we define the object language as well as a big-step interpreter. The `eval` function evaluates a given expression following OCaml's call-by-value, right-to-left strategy. For instance, when given an application `e1 e2`, it first evaluates the argument `e2`, then evaluates the function `e1`. Once the application has been turned into a redex, we perform $\beta$-reduction, and evaluate the post-reduction expression. Note that, when the top-level expression is an executable, closed program, the input of the `eval` function cannot be a variable. The reason is that we never touch a function's body before it receives an argument, and that $\beta$-reduction replaces lambda-bound variables with values.

Object-level exception handling is performed by the meta-level `try` and `raise` constructs. Specifically, when evaluating `raise e`, we first evaluate `e` to some value `v`, and then raise a meta-level (OCaml) exception `Error v`. If an exception `Error v` was raised during evaluation of `e1` in `try e1 with x -> e2`, the `eval` function ignores the rest of the computation in `e1`, and evaluates `e2` with `v` substituted for `x`. This is exactly how OCaml's try-with construct works. For convenience, we will hereafter call `e1` a *tryee*; the intention is that `e1` is the expression being "tried" by the handler.

The main function `start` calls `eval` in an exception handling context. From the construction, we can see that any expression that has a `raise e` with no matching `try` clause will be evaluated to `raise v`. For example, `2 + 3 + (raise 4) + 5` evaluates to `raise 4`.

## 2.2 Turning the Interpreter into a Stepper

As stated in Section 1, a stepper must display the whole program at each reduction step. Consider the simple arithmetic expression `(1 + 2 * 3) + 4`. When step-executing this expression, we want to see the following reduction sequence:

$$\begin{array}{ll} & (1+2*3)+4 \\ \rightarrow & (1+6)+4 \\ \rightarrow & 7+4 \\ \rightarrow & 11 \end{array}$$

The interpreter in Figure 4, however, does not immediately give us these steps. Suppose the `eval` function is evaluating the subexpression `2 * 3`. We can display this subexpression using a printing function, but we do not have enough information to reconstruct the whole program. What is missing here is the *context* surrounding `2 * 3`, namely `(1 + [.]) + 4` (where `[.]` denotes the hole of the context). Hence, to implement a stepper, we need to keep track of every evaluation context we have traversed.

In Figure 5, we define context frames as algebraic data of type `frame_t`. Each frame represents evaluation of some subexpression: *e.g.*, `CAppR (e1)` tells us that we are evaluating the argument part of an application, whose function part is `e1`. Evaluation contexts are defined as lists of these frames

```
(* exception holding the value of input program's exception *)
exception Error of e_t

(* evaluate expression *)
(* eval : e_t -> e_t *)
let rec eval expr = match expr with
  | Var (x) -> failwith ("unbound variable: " ^ x)
  | Fun (x, e) -> Fun (x, e)
  | App (e1, e2) ->
    begin
      let v2 = eval e2 in
      let v1 = eval e1 in
      match v1 with
      | Fun (x, e) ->
        let e' = subst e x v2 in   (* substitute v2 for x in e *)
        let v = eval e' in
        v
      | _ -> failwith "not a function"
    end
  | Try (e1, x, e2) ->
    begin
      try
        let v1 = eval e1 in
        v1
      with Error (v) ->
        let e2' = subst e2 x v in   (* substitute v for x in e2 *)
        eval e2'
    end
  | Raise (e) ->
    let v = eval e in
    raise (Error (v))

(* start evaluation *)
(* start : e_t -> e_t *)
let start e =
  try
    eval e
  with
    Error v -> Raise v
```

Figure 4: Big-step interpreter

```
(* context frames *)
type frame_t =
             | CAppR of e_t           (* e [.] *)
             | CAppL of e_t           (* [.] v  *)
             | CTry of string * e_t   (* try [.] with x -> e *)
             | CRaise                 (* raise [.] *)

(* evaluation contexts *)
type c_t = frame_t list

(* reconstruct the whole program *)
(* plug : e_t -> c_t -> e_t *)
let rec plug expr ctxt = match ctxt with
  | [] -> expr
  | CAppR (e1) :: rest -> plug (App (e1, expr)) rest
  | CAppL (e2) :: rest -> plug (App (expr, e2)) rest
  | CTry (x, e2) :: rest -> plug (Try (expr, x, e2)) rest
  | CRaise :: rest -> plug (Raise expr) rest
```

Figure 5: Contexts and reconstruction function; first attempt

(spoiler alert: this does not work for exceptions). We then define the `plug` function, which reconstructs a program by wrapping the expression `expr` with context frames in `ctxt`.

Now, if we let the evaluation function receive an additional argument representing the context, we should be able to display all the steps of the arithmetic expression `(1 + 2 * 3) + 4`. For instance, when evaluating the subexpression `2 * 3`, the extra argument will be a two-element list `[(1 + [.]); ([.] + 4)]`, and we can obtain the whole program using the `plug` function.

The resulting stepper is essentially the CK abstract machine [4], where the expression is the control string and the evaluation context is the continuation. Substitution is used to implement $\beta$-reduction. We did not implement the abstract machine directly but augmented a big-step interpreter, because we want to keep the correspondence between big-step execution and small-step execution. It enables us to skip evaluation of user-specified function application, as we elaborate in Section 2.3.

Unfortunately, this naïve implementation does not work in the presence of exception handlers. Consider `try (2 + 3 * (raise 4) + 5) with x -> x`. When step-executing this expression, we expect to see the following steps:

```
(* Step 0 *) try ( 2 + (3 * (raise 4)) + 5 ) with x -> x
(* Step 1 *) try ( raise 4 ) with x -> x
(* Step 1 *)  try (raise 4) with x -> x
(* Step 2 *)  4
```

The first reduction happens when the input to the stepping interpreter is `raise 4`. However, observe that the highlighted redex is a bigger expression `(2 + 3 * (raise 4) + 5)`, because reduction of a `raise` construct discards the context *within* the tryee. Since context frames are collected in a single list, the second argument at this point will be `[(3 * [.]); (2 + [.]); ([.] + 5); (try [.] with`

```
(* frames *)
type frame_t =
             | CAppR of e_t          (* e [.] *)
             | CAppL of e_t          (* [.] v *)
             | CRaise                (* raise [.] *)

(* try frame *)
type ctry_t =
            | CHole                          (* [.] *)
            | CTry of string * e_t * c_t     (* try [.] with x -> e *)

(* evaluation context *)
and c_t = frame_t list * ctry_t

(* reconstruct tryee *)
(* plug_in_try : e_t -> frame_t list -> e_t *)
let rec plug_in_try expr ctxt = match ctxt with
  | [] -> expr
  | first :: rest -> match first with
    | CAppR (e1) -> plug_in_try (App (e1, expr)) rest
    | CAppL (e2) -> plug_in_try (App (expr, e2)) rest
    | CRaise -> plug_in_try (Raise (expr)) rest

(* reconstruct the whole program *)
(* plug : e_t -> c_t -> e_t *)
let rec plug expr (clist, tries) =
  let tryee = plug_in_try expr clist in
  match tries with
  | CHole -> tryee
  | CTry (x, e2, outer) -> plug (Try (tryee, x, e2)) outer
```

Figure 6: Contexts and reconstruction function; final version

`x -> x)]`, *i.e.*, it contains the context *outside* the tryee. This suggests that, when dealing with exception handlers, we have to distinguish between contexts inside and outside a tryee.[1]

In Figure 6, we present a refined definition of evaluation contexts. We see a new definition of context frames `frame_t`, where `CTry` is missing. When evaluating a program that uses try-with constructs, these frames are used to build a delimited context within a tryee. We next find a separate datatype `ctry_t`, which can be understood as meta contexts. Then we define evaluation contexts as pairs of delimited and meta contexts. As an example, when evaluating `raise 4` in the following expression:

```
0 + (try 1 + 2 * (try (3 + raise 4) - 5 with x -> x + 6) with y -> y)
```

the current context looks like:

[1] The destination is not necessary, if we want to support only exception handling. We could simply search for the enclosing handler in the evaluation context. However, it requires a linear search through the evaluation context. Furthermore, distinction is necessary if we want to implement more general control operators, such as shift and reset [2].

```
([(3 + [.]); ([.] - 5)],
  CTry ("x", x + 6,
    ([2 * [.]; 1 + [.]],
      CTry ("y", y,
        ([0 + [.]], CHole)))))
```

The refined contexts allow us to first reconstruct the expression up to the tryee using the `frame_t` contexts, and then build up the whole program using the `ctry_t` contexts. In our particular example, the stepper reconstructs `(3 + raise 4) - 5`, highlights it, and reconstructs the whole program.

To give the reader a better idea how context frames are accumulated, let us demonstrate the evaluation of an expression involving exception handling:

```
eval (2 * (try 3 + (raise 4) - 5 with x -> x + 6)) ([], CHole)
eval (try 3 + (raise 4) - 5 with x -> x + 6) ([2 * [.]], CHole)
eval (3 + (raise 4) - 5) ([], CTry ("x", x + 6, ([2 * [.]], CHole)))
eval 5 ([3 + (raise 4) - [.]], CTry ("x", x + 6, ([2 * [.]], CHole)))
eval (3 + (raise 4)) ([[.] - 5], CTry ("x", x + 6, ([2 * [.]], CHole)))
eval (raise 4) ([3 + [.]; [.] - 5], CTry ("x", x + 6, ([2 * [.]], CHole)))
eval 4 ([raise [.]; 3 + [.]; [.] - 5], CTry ("x", x + 6, ([2 * [.]], CHole)))
eval (4 + 6) ([2 * [.]], CHole)
```

Observe that we discard the context within the tryee, namely `3 + (raise [.]) - 5`, at the last step.

Now we present our stepping interpreter in Figure 7. The function extends the big-step interpreter in two ways (as shaded in the figure): (i) it receives an argument representing the evaluation context; and (ii) it outputs the current program every time reduction takes place.

Let us observe the application case. As in the big-step interpreter, we first evaluate `e2`, and then `e1`. When `e1` has reduced to a function, we know that the application is a $\beta$-redex. In the standard interpreter, what we do is to perform the substitution `subst e x v2` and then evaluate the result. In the stepper, on the other hand, we have an additional function call to the `memo` function defined in Figure 8. This function receives three arguments: the redex we have just found, its reduct, and the current evaluation context. When given these arguments, the `memo` function reconstructs and prints the pre- and post-reduction programs, using the `plug` and `print_exp` functions[2]. After printing the programs, we continue evaluation as usual.

In the `eval` function, we find three more occurrences of `memo`, representing the following reduction rules:

- `try v with x -> e` $\rightsquigarrow$ `v`
- `try raise v with x -> e2` $\rightsquigarrow$ `subst e2 x v`
- `... (raise v) ...` $\rightsquigarrow$ `raise v`

Note that, although the second reduction always happens right after the third one, we keep them as separate rules. The reason is that we need the latter to reduce a raise construct with no matching try clause: *e.g.*, `3 + (raise 4) - 5` $\rightsquigarrow$ `raise 4`. Separating the two reductions also has an educational benefit: it clearly tells us that exception handling consists of two tasks: discarding the context and substituting the value.

[2] In the actual implementation, we annotate redexes and reducts using OCaml's *attributes*. Here, we write `green expr1` to mean `expr1[@stepper.redex]`, and similarly for `purple`. When displaying the steps, the Emacs Lisp program uses the attributes information to appropriately highlight expressions.

```
(* stepping evaluator *)
(* eval : e_t -> c_t -> e_t *)
let rec eval expr ctxt = match expr with       (* add an argument for context *)
  | Var (x) -> failwith ("unbound variable: " ^ x)
  | Lam (x, e) -> Lam (x, e)
  | App (e1, e2) ->
    begin
      let v2 = eval e2 (add ctxt (CAppR e1)) in          (* add context info *)
      let v1 = eval e1 (add ctxt (CAppL v2)) in          (* add context info *)
      match v1 with
      | Lam (x, e) ->
        let e' = subst e x v2 in
        memo (App (v1, v2)) e' ctxt;                      (* output programs *)
        let v = eval e' ctxt in                          (* add context info *)
        v
      | _ -> failwith "not a function"
    end
  | Try (e1, x, e2) ->
    begin
      try
        let v1 = eval e1 (add_try ctxt x e2) in          (* add context info *)
        memo (Try (v1, x, e2)) v1 ctxt;                   (* output programs *)
        v1
      with Error (v) ->
        let e2' = subst e2 x v in
        memo (Try (Raise v, x, e2)) e2' ctxt;             (* output programs *)
        eval e2' ctxt                                    (* add context info *)
    end
  | Raise (e0) ->
    let v = eval e0 (add ctxt CRaise) in                 (* add context info *)
    begin match ctxt with
        | ([], _) -> ()
        | (clist, tries) ->
          memo (plug_in_try (Raise v) clist)              (* output programs *)
               (Raise v)
               ([], tries)
    end;
    raise (Error (v))
```

Figure 7: Stepping evaluator

```
(* output programs *)
(* memo : e_t -> e_t -> c_t -> unit *)
let memo expr1 expr2 ctxt =
  print_exp (plug (green expr1) ctxt);
  print_exp (plug (purple expr2) ctxt)

(* start step-evaluation *)
(* start : e_t -> e_t *)
let start e =
  try
    eval e ([], CHole)    (* initial context *)
  with
    Error (v) -> (Raise v)
```

Figure 8: `memo` and main functions

## 2.3 The Actual Stepper

In Figure 9, we show a reduction sequence produced by the actual stepping evaluator. The evaluator supports the following syntactic constructs:

- integers, floating point numbers, booleans, characters, strings
- lists, tuples, records
- user-defined datatypes
- conditionals, let-expressions, recursive functions, pattern-matching
- exception handling operators
- printing functions and sequential execution
- the List module, user-defined modules
- references, arrays

To allow the user to adjust granularity of steps, we provide an option for skipping the evaluation of the current function application. Let us look at Figure 10, which shows skipping of the factorial function. By pressing the "skip" button, we can directly go from the program on the left to the one on the right, without seeing the intermediate steps that appear during the evaluation of the function's body. This feature helps us focus on the steps we are interested in, allowing us to grasp the overall flow of the execution.

The skipping feature requires some modifications to the `eval` function (Figure 11). The idea is to sandwich the steps within an application between two strings: `(* Application n start *)` and `(* Application n end *)`. Here, `n` tells us at which step we have entered the application. These strings are printed using the `apply_start` and `apply_end` functions, and help the Emacs Lisp program to hide unnecessary steps. We show an example output sequence in Figure 12.

```
(* Step 0 *)
(match (12.0, "x") with
  | (f,s) -> if f = 0.0 then "" else (string_of_float f) ^ s)

(* Step 1 *)
(if 12.0 = 0.0 then "" else (string_of_float 12.0) ^ "x")

(* Step 1 *)
if (12.0 = 0.0)
then ""
else (string_of_float 12.0) ^ "x"

(* Step 2 *)
if (false) then "" else (string_of_float 12.0) ^ "x"

(* Step 2 *)
(if false then "" else (string_of_float 12.0) ^ "x")

(* Step 3 *)
((string_of_float 12.0) ^ "x")
```

```
(* Step 3 *)
(string_of_float 12.0) ^ "x"

(* Step 4 *)
("12.0") ^ "x"

(* Step 4 *)
("12.0" ^ "x")

(* Step 5 *)
("12.0x")

(* Step 5 *)
"12.0x"

(* Stepper.go ends. *)
```

Figure 9: Evaluating programs using the actual stepper

```
(* Step 0 *)
let fact3 = (factorial 3)

(* Step 1 *)
let fact3 = (if 3 = 0 then 1 else 3 * (factorial (3 - 1)))
```

```
(* Step 0 *)
let fact3 = (factorial 3)

(* Step 18 *)
let fact3 = (6)
```

Figure 10: Skipping evaluation of the factorial function

```
let rec eval expr ctxt = match expr with
    ...
  | App (e1, e2) ->
    begin
      let v2 = eval e2 (add ctxt (CAppR e1)) in
      let v1 = eval e1 (add ctxt (CAppL v2)) in
      match v1 with
      | Lam (x, e) ->
        let e' = subst e x v2 in
        let apply_num = apply_start () in                (* output start mark *)
        memo (App (v1, v2)) e' ctxt;
        let v = eval e' ctxt in
        apply_end apply_num;                               (* output end mark *)
        v
      | _ -> failwith "not a function"
    end
  | ...
```

Figure 11: Skipping application

```
(* Step 0 *) (f 4) + 10 * 100
(* Step 1 *) (f 4) + 1000
(* Application 1 start *)
(* Step 1 *) f 4 + 1000
(* Step 2 *) (4 * 2) - 1 + 1000
(* Step 2 *) (4 * 2 - 1) + 1000
(* Step 3 *) (8 - 1) + 1000
(* Step 3 *) 8 - 1 + 1000
(* Step 4 *) 7 + 1000
(* Application 1 end *)
(* Step 4 *) 7 + 1000
(* Step 5 *) 1007
```

Figure 12: Stepping application

# 3 Stepping OCaml in the Classroom

Since 2016, we have been using (earlier versions of) our stepper in an introductory OCaml course called "Functional Programming", taught by the third author at Ochanomizu University.

## 3.1 The OCaml Course

The "Functional Programming" course teaches how to program with functions and types, covering basic topics such as recursion, datatypes, effects, and modules. The course consists of 15, weekly lab sessions, and each session consists of 90 minutes lab-style class per week. (Many students remain in the lab after 90 minutes up until around 150 minutes.) Throughout the course, students build a program that searches for the shortest path based on Dijkstra's algorithm. The participants of the course are second-year undergraduate students majoring in computer science (around 40 students each year). All students enter this course after a CS 1 course in the C programming language.

The course is taught in a "flipped classroom" style. Before every meeting, students are asked to study assigned readings and videos prepared by the instructor and answer simple quizzes. In the classroom, they practice the newly covered topics through exercises, with assistance of the instructor as well as five to six teaching assistants (including the first and second authors).

The exercises include simple practice problems and report problems. The former are for confirming students' understanding of the topics and are expected to be completed within a class. The latter problems (for credit) are due in one week. Whenever a student executes a program, by either step execution or standard execution, the program as well as its execution log (syntax errors, type errors, or the result of execution) are recorded.

For most of the problems (up to the 12th week), we provide a check system where students can submit their solutions to see whether they pass the given tests. To earn points for report problems, students are required to have their programs pass the check system.

## 3.2 The Uses of the Stepper

We introduced the stepper into Functional Programming in 2016. Although the steppers used in the past had almost the same user-interface as the one shown in Figure 2, they differ in the following ways:

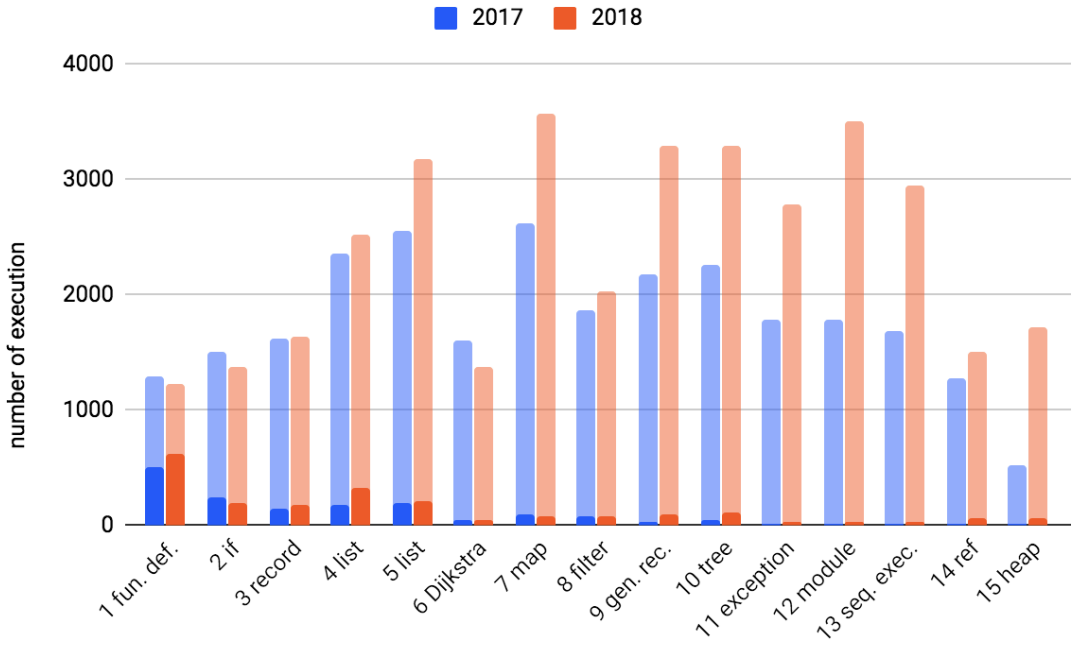


Figure 13: Frequency of standard execution (light-colored) and step execution (dark-colored) in each week in 2017 and 2018. The stepper was not used at all toward the end of the course in 2017, but it was used in some degree in 2018.

**2016** This first version supported function definitions, conditionals, records, lists, and recursion. However, there were various operations that were not supported. As such, the usability of the stepper was low. Moreover, when the instructor introduced the stepper to students, he only mildly encouraged to use it. Although we do not know how much the stepper was used in 2016 since we did not log the execution of stepper, we expect it was used only rarely in the first few weeks of the course.

**2017** Based on the lessons from the previous year, the second version supported most operations used in the first six weeks. The instructor introduced the stepper up front at the first class and showed how to use the stepper with various examples in the subsequent classes.

**2018** The third version supported almost all the constructs needed for the course, including modules, exception handling, sequential execution, printing, references, and arrays. It also supported skipping of function application. The instructor introduced the stepper as though the stepper was the only way to execute OCaml programs, encouraging the uses of the stepper. (Students gradually realized that they could execute a program in the interpreter in a few weeks.)

Figure 13 shows how many times students used the stepper among all the executions including the ones that ended up in an error. In both 2017 and 2018, the stepper was used quite often until week 5. This is partly because we encouraged students to use the stepper when they had trouble finding bugs and understanding recursion. After week 5, the number decreases, because students started using an interpreter, too, as programs became larger.

In 2017, the number of stepper uses decreases toward the end of the course. In contrast, in 2018, certain number of stepper uses is observed, thanks to the support of exception handling, modules, and references. Figure 14 shows the number of execution of programs using these features during step-

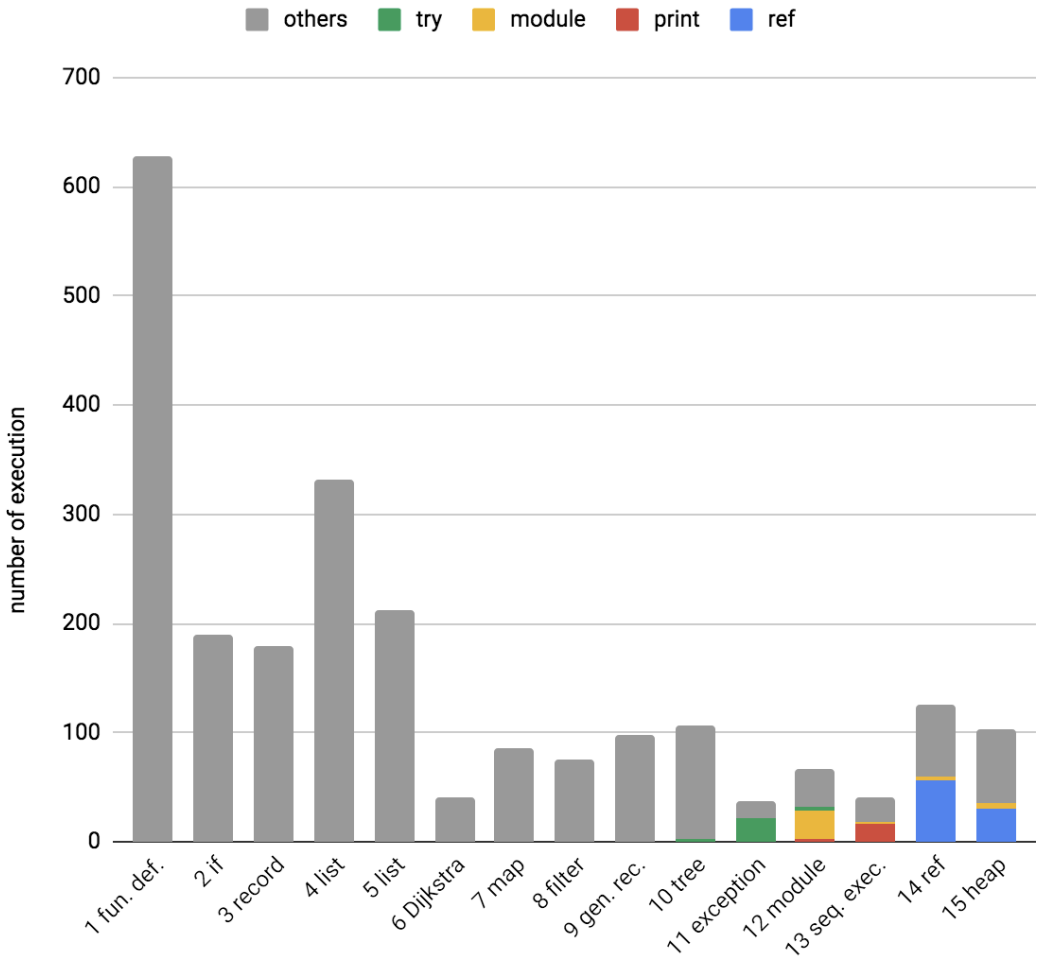


Figure 14: Number of times the stepper was used to evaluate a program with “try”, “module”, “print” or “ref” in 2018.

execution in 2018. From the figure, we can see that there is a demand for step execution of advanced constructs such as exception handling and modules.

The exact numbers of execution are available in Table 2 in the Appendix.

### 3.3 Effects of Stepper

It is not easy to see the effect of a tool like a stepper on the learning of students. In the case of improving error messages of a compiler, for example, one can classify various errors and see how many of them are covered by the improved error messages objectively. For the stepper, it is unclear how to show such numeric data.

As an attempt to measure the effect of the stepper, we examined how long students took to submit correct solutions to the check system. Among all the submitted correct solutions, we gathered the (wall-clock) times of submissions recorded in the check system that are within 100 minutes from the beginning of the class and compared the average times among 2016, 2017, and 2018.

Figure 15 shows the number of questions for which students submitted a correct answer within sig-

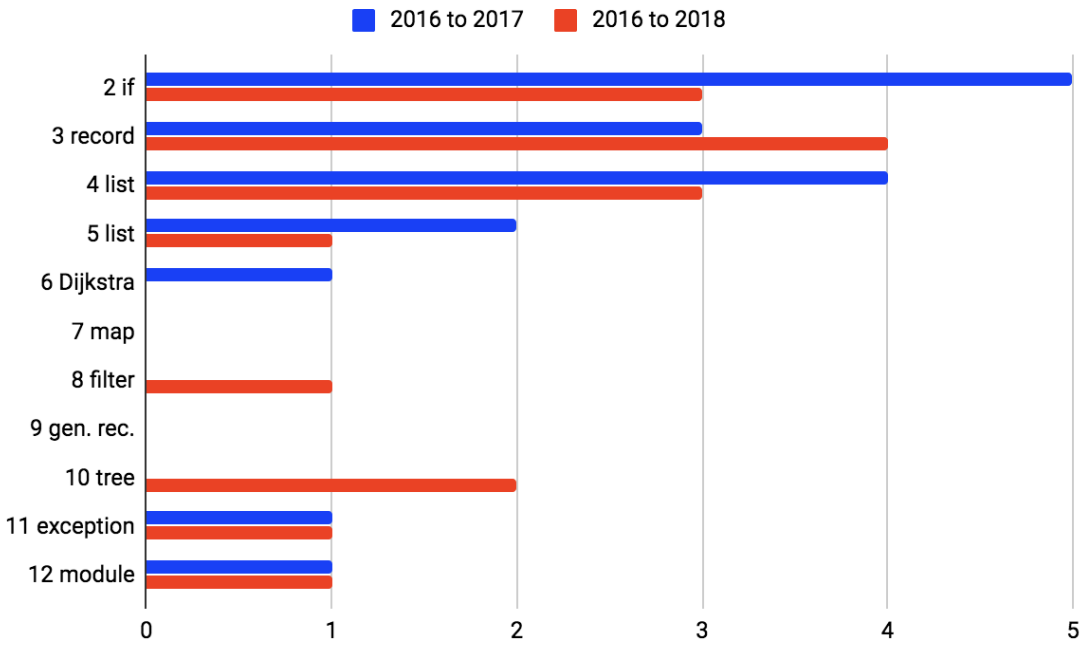


Figure 15: Number of questions where students arrived at a correct answer significantly faster in 2017 and 2018 than 2016.

nificantly shorter time in 2017 and 2018 compared to 2016. The data are based on one-sided t-testing with p-value $< 0.05$; we refer the reader to Table 3 of the appendix for details. Note that we did not include week 1 because we had a special pre-test in the first lecture in 2018.

From the figure, we can see improvement of submission times after the (real) introduction of the stepper, especially in earlier problems. However, there is an exception: for one problem in the 6th week, correct submissions come significantly later in 2018 than in 2016. The problem simply asks students to write a recursive function that adds 1 to each element of a given list. We do not know why it took so long in 2018. The result of t-testing all the problems together is t(1778) = 2.819 (p=0.002) in 2017 and t(2111) = 2.592 (p=0.005) in 2018.

We also compared the average times between 2017 and 2018. For earlier weeks (up to week 5), submissions in 2017 were significantly earlier, while for later weeks, there were no significant difference (except for two problems where the average times for 2018 were earlier). Putting all the problems together, the two were not significantly different with t(1953)=0.455 (p=0.324).

**Threats to validity.** It is possible that the results of our experiment were affected by the enrolled students in each year (there was no over-lapping). In all the three years, the instructor started the class with some introductory comments that vary in length. Although the instructor made similar comments in each year, they were not exactly the same, which could have affected.

## 3.4 Students' Evaluation

At the end of the semester of 2018, we asked the students to share their thoughts on the stepper. We received responses from 38 out of 42 students. We first asked whether the stepper was useful on the 0 to

| | score | # of students |
|---|---|---|
| Using the stepper, I could almost always understand the behavior of programs or the cause of errors. | 4 | 3 |
| Using the stepper, I could often solve problems at hand. | 3 | 8 |
| Using the stepper, I could sometimes solve problems at hand. | 2 | 25 |
| I could rarely find new things using the stepper. | 1 | 2 |
| The stepper was useless. I did not use the stepper. | 0 | 0 |

Table 1: Students' scoring of the stepper in 2018. 38 students out of 42 answered. The average is 2.3.

4 point scale. The results, which we present in Table 1, suggest that the stepper is not a silver bullet that is useful for all the time. However, most students could solve the problems at hand sometimes using the stepper. We are encouraged to see some students choose "the stepper was almost always useful".

We next asked students to write when the stepper was useful (if any), such as when they found their misunderstanding, or when they could deepen their understanding. We summarize the answers in two categories.

**Understanding of the behavior of programs.** Seven students answered they could deepen their understanding of the behavior of programs. In particular, five students among them wrote explicitly that the stepper helped them figure out how functions consume recursive data. We imagine it was particularly instructive to see how a recursive function definition is unfolded in nesting application.

Other students answered that they could observe the behavior of programs in general. They found that arguments of a function are evaluated before the function call, and that the elements of a list are evaluated one by one. Among them, one student observed the right-to-left execution employed in OCaml. Previously, such subtle behavior was taught only in passing without much emphasis.

**Debugging.** Many students found the stepper useful for debugging. Sixteen students answered they could find what was wrong when their program did not pass test cases. By observing each step of execution, they could identify when the program behaved differently from their expectation. This is an important step toward debugging in general. Because printing (and side effects) is handled at the end of the course, the only debugging method for students had been unit testing: they checked whether all the component functions worked as expected. With the stepper, they can simply observe execution of the program and see when it goes wrong.

Three students found the stepper useful to understand why their program did not terminate. Without printing, it is not easy for students to identify the cause of infinite loops. Using the stepper, one of the students could not only observe the infinite loop, but also see how far her program went well and when it went wrong.

## 4 Related Work

Clements et al. [1] describe the implementation of the DrRacket stepper. They point out that when building a stepper, we have to make sure that (i) it displays every element of the reduction sequence in the correct order; and (ii) it has access to information necessary for reconstructing the whole program,

namely evaluation contexts. Based on this idea, they define the stepper as a composition of the following three functions:

- A breakpoint-inserting function, which places a breakpoint to every piece of a program where reduction takes place
- An annotating function, which decorates the user program so that it manipulates evaluation contexts appropriately
- A reconstructing function, which builds the whole program using the context accumulated in the stack

As can be inferred, where they insert breakpoints exactly corresponds to where we insert the `memo` function. Contexts are handled via two primitives of Racket: `w-c-m` ("with-continuation-mark") and `c-c-m` ("current-continuation-marks"). The former can be understood as extending the context list of our stepper, whereas the latter collects all the context frames on the stack. The reconstructing function plays the same role as our `plug` function.

PLT Redex [3] is a domain specific language for formalizing operational semantics. The language provides facilities for defining grammars and reduction rules, and inherits the algebraic stepper from the DrRacket environment [5]. In addition to these, Redex has the ability of generating *reduction graphs* of programs. A reduction graph is essentially a graphical version of a stepper's output, where the elements of a reduction sequence are connected with arrows. Reduction graphs are more informative in that each arrow is annotated with the reduction rule used in that step.

Tunnel Wilson et al. [6] investigate students' understanding of functions and recursion via tracing activities. Instead of adopting the traditional tracing method that uses stacks, they take a substitution-based approach, where students rewrite programs using a set of reduction rules. The rewriting activity can be viewed as writing down the output of a stepper by hand, although their purpose was not to assist debugging.

## 5 Conclusion

In this paper, we presented an OCaml stepper that supports advanced features including exception handling. The main idea is to keep track of two levels of evaluation contexts, which we use to reconstruct pre- and post-reduction programs. We also shared students' feedback on our stepper. Although it is not easy to measure the effectiveness of the stepper in education, we reported that the stepper is used by many students with positive reaction.

The "Functional Programming" course is taught every year, which means we are constantly having new users of our stepper. As future work, we intend to find other ways to assess the stepper, and further evaluate its impact on students. We believe that this would provide new insight into the development and assessment of pedagogical tool in general.

## Acknowledgments

We are grateful to anonymous reviewers for constructive comments and criticisms. This work was partly supported by JSPS KAKENHI under Grant No. 15K00090.

## References


[1] John Clements, Matthew Flatt & Matthias Felleisen (2001): *Modeling an algebraic stepper*. In: *European symposium on programming*, Springer, pp. 320–334, doi:10.1007/3-540-45309-1_21.

[2] Olivier Danvy & Andrzej Filinski (1990): *Abstracting Control*. In: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, LFP '90, pp. 151–160, doi:10.1145/91556.91622.

[3] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics engineering with PLT Redex*. MIT Press.

[4] Matthias Felleisen & Daniel P. Friedman (1986): *Control operators, the SECD-machine, and the λ-calculus*. In M. Wirsing, editor: *Formal Description of Programming Concepts III*, Elsevier, pp. 193–219. Available at https://cs.indiana.edu/ftp/techreports/TR197.pdf.

[5] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt & Robert Bruce Findler (2012): *Run Your Research: On the Effectiveness of Lightweight Mechanization*. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/2103656.2103691.

[6] Preston Tunnell Wilson, Kathi Fisler & Shriram Krishnamurthi (2018): *Evaluating the Tracing of Recursion in the Substitution Notional Machine*. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, ACM, pp. 1023–1028, doi:10.1145/3159450.3159479.


## A Appendix

| | 2017 | | | | | | 2018 | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| week | all | step. | try | mod. | print | ref | all | step. | try | mod. | print | ref | contents |
| 1 | 1293 | 504 | 0 | 0 | 0 | 0 | 1233 | 627 | 0 | 0 | 0 | 0 | fun. def. |
| 2 | 1511 | 235 | 0 | 0 | 0 | 0 | 1375 | 189 | 0 | 0 | 0 | 0 | if |
| 3 | 1618 | 144 | 0 | 0 | 0 | 0 | 1641 | 179 | 0 | 0 | 0 | 0 | record |
| 4 | 2364 | 169 | 0 | 0 | 0 | 0 | 2517 | 332 | 0 | 0 | 0 | 0 | list |
| 5 | 2556 | 193 | 0 | 0 | 0 | 0 | 3173 | 213 | 0 | 0 | 0 | 0 | list 2 |
| 6 | 1596 | 43 | 0 | 0 | 0 | 0 | 1369 | 41 | 0 | 0 | 0 | 0 | Dijkstra |
| 7 | 2621 | 92 | 0 | 0 | 0 | 0 | 3570 | 86 | 0 | 0 | 0 | 0 | map |
| 8 | 1874 | 81 | 0 | 0 | 0 | 0 | 2028 | 75 | 0 | 0 | 0 | 0 | filter |
| 9 | 2184 | 34 | 0 | 0 | 0 | 0 | 3300 | 98 | 0 | 0 | 0 | 0 | gen. rec. |
| 10 | 2254 | 48 | 0 | 0 | 0 | 0 | 3298 | 106 | 3 | 0 | 0 | 0 | tree |
| 11 | 1783 | 20 | 10 | 0 | 0 | 0 | 2790 | 37 | 22 | 0 | 0 | 0 | exception |
| 12 | 1785 | 12 | 8 | 4 | 0 | 0 | 3501 | 37 | 3 | 26 | 3 | 0 | module |
| 13 | 1678 | 10 | 0 | 7 | 2 | 0 | 2943 | 22 | 0 | 3 | 16 | 0 | seq. exec. |
| 14 | 1280 | 11 | 0 | 0 | 0 | 1 | 1511 | 65 | 0 | 4 | 0 | 56 | ref |
| 15 | 517 | 6 | 0 | 0 | 0 | 0 | 1717 | 68 | 0 | 5 | 0 | 30 | heap |

Table 2: Number of uses of the stepper (step.) among all the executions (all) in 2017 and 2018. The columns try, mod., print, and ref represent number of uses of the stepper for programs that contain exception handling, modules, printing (and sequential execution), and references (including arrays), respectively. The rightmost column shows representative topics handled in the week.

| week. problem | 2016 to 2017 t | p | +/- | 2016 to 2018 t | p | +/- | contents |
|---|---|---|---|---|---|---|---|
| 2.r1 | t(55)=2.098 | p=0.020 | dec | t(60)=0.635 | p=0.264 | dec | if |
| 2.r2 | t(56)=2.364 | p=0.011 | dec | t(57)=1.831 | p=0.036 | dec | |
| 2.r3 | t(54)=1.896 | p=0.032 | dec | t(59)=0.751 | p=0.228 | dec | |
| 2.1 | t(66)=3.006 | p=0.002 | dec | t(74)=3.372 | p=0.001 | dec | |
| 2.2 | t(56)=3.672 | p=0.000 | dec | t(62)=3.036 | p=0.002 | dec | |
| 3.r1 | t(52)=3.222 | p=0.001 | dec | t(61)=2.936 | p=0.002 | dec | record |
| 3.r2 | t(42)=2.339 | p=0.012 | dec | t(56)=3.467 | p=0.001 | dec | |
| 3.r3 | t(41)=1.373 | p=0.089 | dec | t(51)=2.688 | p=0.005 | dec | |
| 3.1 | t(28)=5.610 | p=0.000 | dec | t(38)=2.753 | p=0.004 | dec | |
| 3.2 | t(17)=1.655 | p=0.058 | dec | t(27)=0.105 | p=0.459 | dec | |
| 3.3 | t(16)=1.546 | p=0.071 | dec | t(13)=0.603 | p=0.279 | dec | |
| 4.r1 | t(47)=2.088 | p=0.021 | dec | t(61)=2.446 | p=0.009 | dec | list |
| 4.r2 | t(48)=1.909 | p=0.031 | dec | t(60)=2.267 | p=0.014 | dec | |
| 4.1 | t(51)=2.134 | p=0.019 | dec | t(60)=2.473 | p=0.008 | dec | |
| 4.2 | t(18)=3.033 | p=0.004 | dec | t(20)=0.489 | p=0.315 | dec | |
| 5.r1 | t(42)=1.037 | p=0.153 | dec | t(55)=0.257 | p=0.399 | inc | list 2 |
| 5.1 | t(49)=1.592 | p=0.059 | dec | t(61)=0.904 | p=0.185 | dec | |
| 5.2 | t(55)=4.138 | p=0.000 | dec | t(62)=1.631 | p=0.054 | dec | |
| 5.3 | t(47)=3.305 | p=0.001 | dec | t(50)=1.940 | p=0.029 | dec | |
| 6.r1 | t(30)=0.322 | p=0.375 | inc | t(51)=2.011 | p=0.025 | inc | Dijkstra's algorithm |
| 6.1 | t(41)=1.678 | p=0.050 | dec | t(61)=1.155 | p=0.126 | dec | |
| 6.2 | t(45)=1.415 | p=0.082 | dec | t(62)=0.976 | p=0.166 | dec | |
| 6.3 | t(34)=2.296 | p=0.014 | dec | t(42)=0.548 | p=0.293 | dec | |
| 7.r1 | t(42)=0.462 | p=0.323 | inc | t(56)=0.314 | p=0.377 | dec | map |
| 7.r2 | t(41)=0.286 | p=0.388 | inc | t(54)=1.181 | p=0.121 | dec | |
| 7.r3 | t(40)=0.677 | p=0.251 | inc | t(51)=1.492 | p=0.071 | dec | |
| 7.1 | t(21)=0.965 | p=0.173 | dec | t(20)=0.372 | p=0.357 | dec | |
| 7.2 | t(12)=0.380 | p=0.355 | inc | t(7)=0.686 | p=0.258 | dec | |
| 8.r1 | t(46)=1.162 | p=0.126 | inc | t(58)=2.694 | p=0.005 | dec | filter |
| 8.1 | t(16)=0.844 | p=0.205 | dec | t(22)=0.841 | p=0.205 | dec | |
| 9.r1 | t(44)=1.294 | p=0.101 | dec | t(55)=0.678 | p=0.250 | dec | general recursion |
| 10.r1 | t(48)=0.312 | p=0.378 | dec | t(51)=1.308 | p=0.098 | dec | tree |
| 10.r2 | t(48)=0.457 | p=0.325 | dec | t(50)=1.760 | p=0.042 | dec | |
| 10.1 | t(33)=0.976 | p=0.168 | dec | t(38)=0.845 | p=0.202 | dec | |
| 10.2 | t(19)=1.498 | p=0.075 | dec | t(19)=0.871 | p=0.197 | dec | |
| 10.3 | t(15)=1.538 | p=0.072 | dec | t(12)=2.240 | p=0.022 | dec | |
| 11.r1 | t(43)=0.272 | p=0.393 | dec | t(53)=1.555 | p=0.063 | dec | exception |
| 11.r2 | t(37)=0.454 | p=0.326 | dec | t(44)=1.906 | p=0.032 | dec | |
| 11.r3 | t(31)=0.050 | p=0.480 | inc | t(40)=1.208 | p=0.117 | dec | |
| 11.1 | t(34)=1.567 | p=0.063 | dec | t(46)=1.518 | p=0.068 | dec | |
| 11.2 | t(28)=2.204 | p=0.018 | dec | t(36)=1.563 | p=0.063 | dec | |
| 11.3 | t(17)=0.604 | p=0.277 | dec | t(21)=0.384 | p=0.352 | dec | |
| 12.r1 | t(39)=2.229 | p=0.016 | dec | t(52)=4.009 | p=0.000 | dec | module |
| all | t(1778)=2.819 | p=0.002 | dec | t(2111)=2.592 | p=0.005 | dec | |

Table 3: Result of one-sided t-test with p-values comparing the time between the beginning of the class and the moment that students submitted a correct solution. The column +/- shows whether the average time increased or decreased. The p-values below 0.05 are colored.