Cross validation of the universe teachpack of Racket in OCaml

Chihiro Uehara
Department of Information Science
Ochanomizu University
Tokyo, Japan
uehara.chihiro@is.ocha.ac.jp

Kenichi Asai
Department of Information Science
Ochanomizu University
Tokyo, Japan
asai@is.ocha.ac.jp

The Racket language provides general functions to make interactive games easily in the universe teachpack. The purpose of this study is to verify the effectiveness of interactive game programming pursued in Racket in OCaml. For that purpose, we implemented the same library as the universe teachpack in the statically-typed language OCaml. We performed a user testing on 15 students in our university. The testing shows that we could manage almost the same programming experience in OCaml as in Racket. On the other hand, error messages of the library we made are sometimes confusing and there is room for improvement.

1 Introduction

The design recipe, as advocated in the book *How to Design Programs* (HtDP) [3], teaches us the essential part of programming, i.e., how we should write or *design* programs. Rather than teaching a particular programming language, it provides us with a general guideline for writing programs. It was originally used in the university courses, but has widely spread ever since, and is now used in K-12 education, too. The design recipe is often introduced with the Racket language, but the concept of the design recipe is independent of the programming language used. In fact, the second author has used the design recipe to teach OCaml for more than ten years [1].

On the other hand, the use of graphics and interactive animation/games in programming is becoming popular. Students can not only see and understand the result of programming in graphics, but also *enjoy* it [6] . Accordingly, the second edition of *How to Design Programs* (HtDP2e) [5] uses graphics and animation from the day one. One of the key successes of HtDP2e is its use of the universe framework [4] as implemented in the universe teachpack in Racket. The universe teachpack allows students to use graphics and animation without *any* complicated procedures. Thus, students can concentrate on the essential part of programming with the design recipe, while enjoying graphics and animation. *The design recipe and the universe framework provide an ideal platform for learning essentials of programming.*

However, such an ideal platform exists only in Racket so far. One can use graphics in OCaml, too, but with a lot of complicated procedures that are irrelevant for beginning students. Thus, it is impractical to use graphics in the introductory OCaml course. To use graphics in the introductory courses, it is essential that the interface to graphics is extremely simple and intuitive. Apparently, the simple interface of the universe framework is not fully appreciated outside the community. This is a sad situation, because OCaml also has some advantages as a student language: it consists of relatively small language constructs and is easy to learn; it can describe simple input/output behavior of functions as types (although types are not as strong as contracts [7]); and types are statically checked before the program runs, forcing students to think the input/output behavior rigorously. It also gives us false feeling that the universe framework works well only with Racket.

Submitted to:
TFPIE 2015

To show that the universe framework does not depend on Racket, we ported the universe teachpack of Racket into OCaml, enlarging the applicability of the universe framework. The library we implemented is called the universe library and offers the same functionality as the Racket universe teachpack. Theoretically, any Turing-complete language can simulate any programs. However, it is not at all obvious the implemented library works well in education, even if the original teachpack works well, because the two languages and their programming environments are different. In this paper, we report on our experience of using the universe library of OCaml. This work is the cross validation of the universe framework in OCaml. Specifically, we conducted user testing to confirm that the library is (to certain extent) equally useful in OCaml as in Racket. On the other hand, we also report that there is room for improvements, especially in the error messages of the library.

The paper is organized as follows. In Section 2, we introduce the universe library and its underlying framework. Section 3 briefly shows the implementation of the universe library. After describing the overview of the user testing and created games in Section 4 , we list questions and answers to the students in Section 5. Section 6 discusses the result of the user testing, while Section 7 considers more technical aspects how the universe framework fits in OCaml. We describe related work in Section 8 and the paper conclusion in Section 9.

The library we implemented is called the universe library and is available from
`http://pllab.is.ocha.ac.jp/~asai/Universe/`.

2 The universe library

The central idea of the universe framework is to identify states of a game and to regard interaction as transition from old states to new states. For stand-alone games, the state is called *world*. It consists of all the information needed to specify the state of the game uniquely. The game is then constructed in the *world-passing style*: we provide a draw function that creates a game screen from the world, event handlers that return a new world according to the current world and events (such as key strokes and mouse click), etc. Since calculating a new world from the old world (possibly with additional information such as which key was pressed) does not involve mutation, the world-passing style goes well with the introductory courses: students need to know only the function definition and basic algebra, which are the main initial focus of the introductory programming courses.

In the following, we describe the universe framework using our universe library in OCaml. As an example, we show how to write a simple ball game depicted in Figure 1. It is a two-person game where the goal of the game is to delete all the opponent's moving balls by clicking them. We first describe the one-person version (where the player deletes all the balls of her own) and extend it to the two-person version in the later sections.

2.1 A stand-alone (client) program

The first step is to identify what constitutes the world of the game. In the ball game, the state of the game is uniquely determined by a list of all the balls. We thus define:

```
(* type of world *)
type world_t = ball_t list      (* a list of my balls *)
```

where `ball_t` is a type of balls consisting of the coordinates, radius, and color of a ball. To move balls, we specify how the world changes after the clock ticks as a function from the old world to the new world.

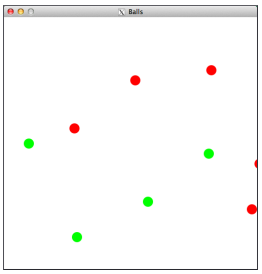

Figure 1: A simple ball game

Assuming that we have defined a function called `move_ball_on_tick` that returns a new ball after the tick from the old ball, we can define the `move_on_tick` function as follows:

```
(* function called at every time interval *)
(* move_on_tick : world_t -> (world_t, 'a) World.t *)
let move_on_tick world =
  let new_world = List.map move_ball_on_tick world in
  World new_world
```

At every time interval, this function is called to update the current world. The game screen is accordingly redrawn using the new world, which moves the balls on the screen. The type `('a, 'b) World.t` is defined in the `World` module of the universe library as follows:

```
type ('a, 'b) t = World of 'a
                | Package of 'a * 'b
```

In communicating games, one can specify a message of type `'b` to be sent to other processes in addition to the new world of type `'a`. In stand-alone games, only the `World` constructor is used.

Once all the necessary functions are defined, the game starts by executing `big_bang`.

```
(* register necessary definitions to big_bang, and start *)
let _ =
  big_bang initial_world          (* initial value of world *)
           ~name:"Ball Game"      (* name of screen *)
           ~to_draw:draw          (* draws a game screen along world *)
           ~width:width           (* width of the screen *)
           ~height:height         (* height of the screen *)
           ~on_mouse:handle_mouse (* called on mouse click *)
           ~on_tick:move_on_tick  (* called at every time interval *)
           ~rate:0.1              (* time interval to call on_tick *)
           ~stop_when:game_over   (* checks whether the game is over *)
```

We use labeled arguments to register necessary values and functions. With labeled arguments, one can register only necessary functions in any order. When omitted, a default function that does nothing is used.

2.2 Communicating client programs

The universe framework allows us to write communicating games. In communicating games, each client program has its own world and they communicate by sending messages through a server. The basic

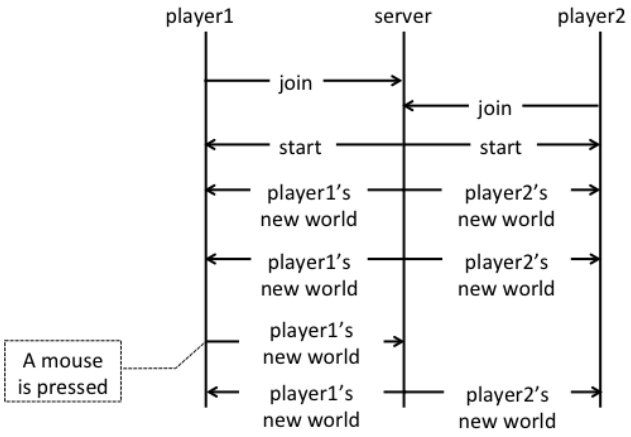

Figure 2: The state of communication of the ball game

structure of the client program is the same as in the stand-alone game. In addition, the client program can now send and receive messages.

To coordinate between clients and a server, it is recommended to write a communication diagram [10]. Figure 2 shows a communication diagram for the ball game. When two clients join, the server sends a start message to start the game.

To turn the stand-alone ball game to a communicating ball game, we first update the definition of world as follows:

```
(* type of the world *)
type world_t = ball_t list      (* a list of my balls *)
             * ball_t list      (* a list of opponent's balls *)
```

To determine the current state of the communicating ball game, we need the information not only of the player's balls but also of the opponent's balls. We accordingly change the draw function so that the opponent's balls are also shown in the screen.

In the stand-alone ball game, balls are moved according to the tick event. In the communicating game, the situation is more complicated than it first appears, because we need to synchronize the time of the two worlds. As reported by Morazan [10], keeping time in the server rather than in clients simplifies the structure of the game. We remove the function `move_on_tick` from the client and handle the tick event in the server (described in the next section). At every time interval, a message is sent from the server consisting of the new world after the tick. (See Figure 2.) To receive a message, we define the function `receive` which receives the old world and a message from the server, and returns a new world. For the ball game, the client simply throws away the old world and uses the received world as a new world:

```
(* function called when a message is received *)
(* receive : world_t -> world_t -> (world_t, 'a) World.t *)
let receive world message = World message
```

We also handle the mouse event in the server. When a mouse is clicked, rather than creating a new world after the mouse click, the clicked coordinates are sent to the server as a message to be handled in the server.

```
(* function called on mouse click *)
(* handle_mouse : world_t -> int -> int -> string -> (world_t, int * int) World.t *)
let handle_mouse world x y event =
  if event = "button_down" then Package (world, (x, y))
                           else World world
```

Finally, we start the client with `big_bang`:

```
(* register necessary definitions to big_bang, and start *)
let _ =
  big_bang initial_world          (* initial value of world *)
           ~name:"Ball Game"      (* name of screen *)
           ~to_draw:draw          (* draws a game screen along world *)
           ~width:width           (* width of the screen *)
           ~height:height         (* height of the screen *)
           ~on_mouse:handle_mouse (* called on mouse click *)
           ~on_receive:receive    (* receives a message from a server *)
           ~stop_when:game_over   (* checks whether the game is over *)
```

2.3 A Server program

In addition to client programs, a communicating game requires a server that governs overall progress of the game. Like the world in a client program, the server keeps track of the state of all the world, or *universe*. The state of the server for the ball game can be defined as a list of pairs of a client identifier (of type `iworld_t`) and a list of balls of the client.

```
(* type of the state *)
type state_t = (iworld_t * ball_t list) list
```

In response to clock ticks, the server moves balls of all the worlds:

```
(* function called at every time interval *)
(* move_on_tick : state_t -> (state_t, world_t) Universe.t *)
let move_on_tick state =
  let new_state = List.map (fun (world_id, lob) ->
                            (world_id, List.map move_ball_on_tick lob))
                           state in
  send_messages new_state
```

The function `send_messages` sends world to all the clients (not shown). Similarly to `('a, 'b) World.t`, the type `('a, 'b) Universe.t` is defined in the `Universe` module where `'a` represents the type of states and `'b` the type of messages sent to clients.

After defining a message handler (to handle mouse events of the clients), we can start the server using `universe`:

```
(* register necessary definitions to universe, and start *)
let _ = universe init_state              (* initial value of state *)
                 ~on_new:onnew           (* called when a client joins *)
                 ~on_tick:move_on_tick   (* called at every time interval *)
                 ~on_msg:change_on_mouse (* called when a message is received *)
                 ~on_disconnect:change_on_disconnect
                                         (* called when a client has left *)
                 ~rate:0.1               (* time interval to call on_tick *)
```

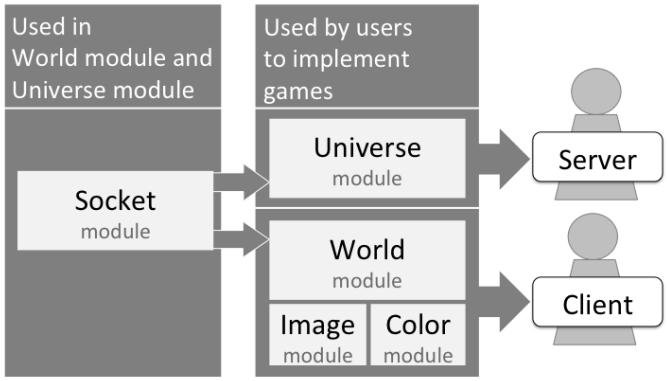

Figure 3: The relationship of modules of the universe library

3 Implementation of the universe library

The universe library consists of five modules.

`World` **module** used to implement client programs; provides `big_bang`.

`Color` **module** provides colors.

`Image` **module** provides image related functions, such as rectangles, polygons, texts, and images.

`Universe` **module** used to implement server programs; provides `universe`.

`Socket` **module** provides low-level functions for interprocess communication.

Using the universe library, one can write both one-player games and many-player games. The first three modules, `World`, `Color`, and `Image`, are used to implement client programs, while the module `Universe` is used to implement server programs. The module `Socket` is used in the modules `World` and `Universe`, but is hidden from the users. The relationship between these modules are depicted in Figure 3.

Images and handling of events (key strokes, mouse events, and timer events) are implemented using LablGtk2, the OCaml interface to GTK+. The `Socket` module is implemented using the Unix module in the standard library.

3.1 Difference from the original universe teachpack of Racket

Apart from the static typing, the interface of the universe library is almost the same as that of the universe teachpack of Racket. There are two differences users observe when they use the universe library.

First, while Racket's integrated environment allows us to paste images into the program, we have no such environment for OCaml. Instead, users have to read images from a file, using `read_image "file.png"`.

Secondly, the universe library uses labeled arguments for `big_bang` and `universe`, while Racket provides a special syntax for them. Although students in the user testing did not know labeled arguments, they had no problem using them.

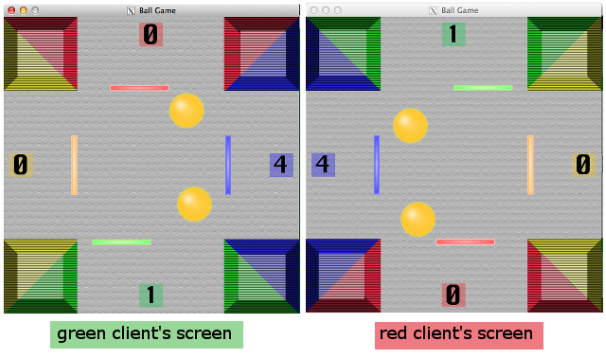

Figure 4: The game screen ot the hockey game

4 The user testing

To see the usability of the universe library, we held a class to create communicating games using the universe library. 15 students enrolled the class and they formed six teams each consisting of two or three students. Among the 15 students, two 1st-year students had basic programming experience only in the C language, while ten 2nd-year students and three 3rd-year students had additional OCaml programming experience (roughly corresponding to the CS1 course without the universe library). Six of the students had programmed non-communicating games in Racket using the universe teachpack in the LA course targeted on non-CS-major students.

Below, we briefly describe the games created by the six teams.

4.1 Game created by the 3rd-year students

The team consisting of three 3rd-year students created a four-person hockey game. Figure 4 shows two screens displayed in two computers of two players (among four). Balls are bounced using the player's bottom-most bar controlled by left and right arrow keys. The balls are accelerated using up arrow key when the balls bounce. The color of the bounced ball is changed to the player's color. When a ball enters one of the player's positions, the player loses one point and the player with the ball's color obtains one point. A player with five points wins. The number of balls starts from one at the beginning of the game, but it increases as the game proceeds.

In Figure 4, we see that the green player obtained one point and the purple player is one more point to win. Player's bar is always displayed at the bottom of the screen. The two balls have just bounced back from the yellow bar (because they are yellow).

The flow of communication of the hockey game is almost the same as that of the ball game (partly because we showed the ball game as an example in the class). See Figure 5. The server controls tick events to synchronize the four clients. The server maintains the world of all the clients as a state, and sends the world to each client at every time interval. The clients send the server a message whenever a key is pressed. The server then creates a new world for each client and send it to each client. The clients update their world as they receive a new world from the server.

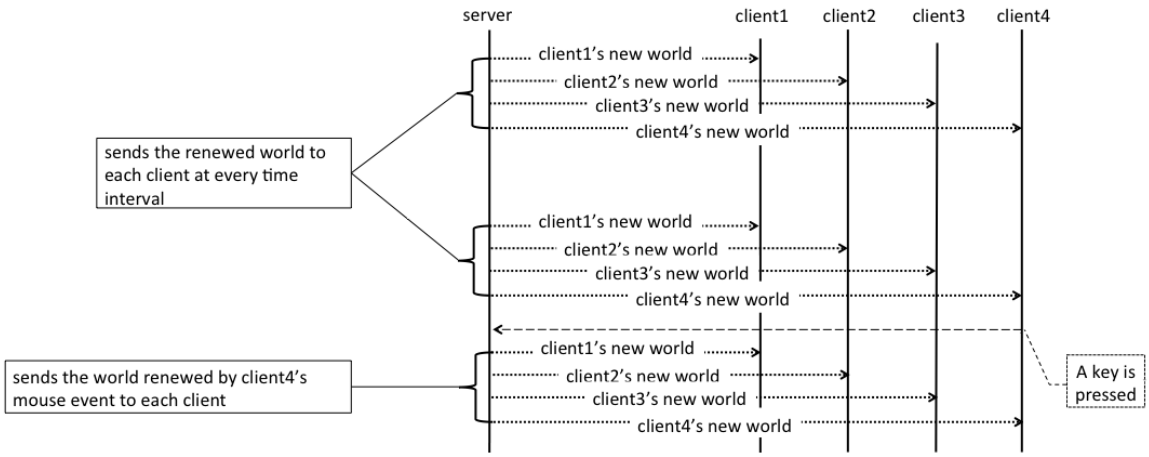

Figure 5: The flow of communication in the game

4.2 Games created by other teams

Four teams consisting of 2nd-year students created three-person old maid card game, two-person checker-like board game called Geister®, two-person shooting game, and three-person Spase Invaders®. All the games were attractive and enjoyable.

The team consisting of two 1st-year students created a simple tag game, with any number of players and one tagger. All the players are controlled by arrow keys. The server is responsible for moving all the players and a tagger. Although the quality of the game was lower than the other teams, they could actually come up with a communicating game that works.

Since most of the enrolled students have already finished CS1, we did not teach the proper CS1 material, but concentrated on the use of the universe library. Thus, the fact that the two 1st-years students without OCaml experience could complete a working game was a great achievement for them. From our perspective, we consider it as a sign that the universe library (and its underlying framework) is simple enough to use for students without prior knowledge in OCaml.

5 Questions to students after the user testing

After the class finished, we asked students for comments on the universe library. This section summarizes questions and answers we obtained from the students. The number in the parenthesis after the comments represents the number of students who made the corresponding comments. All the answers were given in free format and multiple answers were allowed. We discuss the results in the following sections.

5.1 Advantages of the universe library

We first asked the advantages of using the universe library.

1. One can use the universe library without understanding the internals (implementation) of the modules. (10)
2. The universe library provides an easy environment for programming where one needs to think about the world only. (3)

3. The universe library supports first-class images where png images with alpha values can be passed and stored. (2)

5.2 Shortcomings of the universe library

We then asked the shortcomings of the universe library.

1. Error messages are hard to understand. (4)
2. I did not understand the concept of world and state after all. (2)
3. No support for sounds. (2)

5.3 Comparison to the universe teachpack of Racket

Six of the students had experience of using the universe teachpack of Racket. We asked how they compare the universe library with the universe teachpack.

1. The interface is almost the same. (4)
2. Since I am more familiar with OCaml, the universe library was easier to use. (2)
3. It is hard both to fix the type of world and state at first and to change them later. (2)
4. The universe teachpack of Racket is better in that it allows us to embed images directly into programs. (2)

5.4 Other comments

1. Because it was difficult to write test cases for games, I was uneasy if my program was OK until I saw the game screen. (1)
2. I feel more confident in OCaml programming after creating something I can see on the screen. (1)
3. It was difficult to synchronize time and values. (1)
4. I was glad to see that my idea was transcribed into a machine as a program. (1)

6 Results of user testing

In this section, we summarize the overall reaction to the universe library observed from the user testing. More technical issues are discussed in the next section.

6.1 What turned out to be good

As observed in Section 5.4, some students find programming with the universe library attractive, as they can see the result of what they program in the screen. Such an experience was possible because of the interactive game programming provided by the universe framework. During the class, we observed that students enjoy programming in general, with a lot of discussion with team mates. They also experienced pleasure of achievement, joy of completing games they design. The complaints that the universe library has no support for sounds came from their desire to make better games.

Before the class starts, we feared that students might not be able to follow the course to the end, and in particular that the two 1st-year students without prior knowledge on OCaml would find the course too

difficult. It turned out that we did not have to worry. The 15 students decided to enroll in the class in the second lecture and they all finished the class successfully, obtaining their credits. We consider one of the big reasons for their enthusiasm is the attractive environment provided by the universe library.

6.2 What students find difficult

Some students found that synchronization of time and values is difficult. We consider it as a good thing, because at some point, they need to understand synchronization to make communicating games. By struggling with synchronization problem, the students must have deepen their understanding of synchronization. On the other hand, some game turned out to maintain time events in each client independently and failed to synchronize time properly. It suggests that while the universe framework is a good platform to encourage students to think about synchronization, it does not necessarily lead to a uniform solution. Morazan [10] reported a similar observation.

Some students, to our disappointment, did not fully understand the concept of world and state. Since the concept is so natural to us , we did not explain it in great detail, but simply gave a few example games with brief explanation. In retrospect, we should have spent some more time on the universe model itself, before diving into programming.

6.3 Universe library in education

Among 15 students in the user testing, 13 students had already finished the introductory OCaml course in the preceding semester. With the basic knowledge on the design recipe in OCaml, they found programming in the universe library interesting and took part in the course enthusiastically and with success. It shows that the universe library is suitable for the CS2 course. The situation is similar to Morazan [10] where the universe framework was used to program advanced distributed games.

As for the other two students, the course was their first experience to program in OCaml. Without any knowledge on the design recipe in general, it was not easy for them to write interactive programs in OCaml. Yet, they could come up with a working communicating game. It shows that the universe library is easy enough for beginning students to use. We have not yet incorporated the universe library into our introductory OCaml course, but the library appears to be ready to be included in it.

7 How the universe framework fits in OCaml

In this section, we discuss how the programming environment provided by the universe framework fits in OCaml. We consider the influence of static typing, debugging, testing, and error messages in detail.

7.1 Influence of static typing

Whenever we change definition of the world or the state, we need to make changes to all the places where the world or the state is mentioned. With dynamic typing, failure to update all the necessary places leads to runtime errors. To help finding such errors, Racket provides a special mechanism to check the shape of the world at runtime . With static typing, they are detected as type errors.

Some students commented that it is hard to fix the type of world and state. Static typing does not help here: regardless of the language, Racket or OCaml, students struggle to come up with a suitable definition of world and state. Static typing does help searching for the necessary changes when the type

of world and state change. If they forget to change some of them, the static type system reports them as type errors.

7.2 Debugging

Racket has an algorithmic stepper [2] that shows how a program is evaluated step by step. One can even use the stepper for interactive games. If one set up a tick event at every one second, the stepper shows the traces for the tick event at every one second. A stepper is a powerful tool for debugging, to see what went wrong. In addition, Racket can also show the current world in a separate window to see how world changes according to various events.

In OCaml, we don't have a stepper. To show the contents of world, one needs to write explicitly printing statements in a program. Considering that such printing statements require sequential execution that is typically covered only at the end of introductory functional language courses, it is desirable to support such functionality in the universe library.

On the other hand, OCaml comes with a type debugger [12] . We have used it in the introductory OCaml course in our university successfully [8]. Whenever a type error occurs, the type debugger asks questions to the user. By answering these questions, the type debugger leads the user to the source of the type error. The presence of the universe library does not affect the type debugger. Implementation of all the functions provided by the universe library is hidden and only their interface is public. By regarding them as primitives, one can use the type debugger as before.

7.3 Testing

The design recipe emphasizes the importance of testing. In our OCaml introductory course, we instruct students to write basic test cases of the form: `let test = (`*program* `=` *result*`)`, where students specify a program to be executed and its expected result. Using this form, students can write test cases for functions to be registered to `big_bang` and `universe`. However, we have not come up with a test method to check whether interactive games are running as specified.

When testing, the Racket integrated environment displays the test coverage by highlighting program parts that are not executed during the test. Although there exists a coverage tool for OCaml [1], we have not tried it yet with the universe library.

7.4 Error messages

During programming, readable error messages are important to understand what is going on in a program, especially for beginning students. Although we can use the type debugger with the universe library for type errors, we have no support for runtime errors. In this section, we list various runtime error messages and discuss how they can be improved.

7.4.1 No specified image file

When a program tries to read an image file that does not exist, the following error is raised.

```
Fatal error: exception Glib.GError
  ("Failed to open file '*filename*': No such file or directory")
```

[1] http://bisect.x9c.fr/

We consider this message is understandable because it contains enough information on what happened. For beginning students, however, it is desirable not to mention `Glib.GError`, because they have no idea what it is.

7.4.2 Exception in the call-back functions

When a call-back function (that handles tick, key, and mouse events) raises an exception, the followig error message is displayed:

```
In callback for signal *signal name*, uncaught exception: *exception name*
```

where `*exception name*` is the name of the raised exception and `*signal name*` is the name of the signal that is being handled. This error message is not informative enough, because it does not show the name of the user-defined function that raised the exception, but rather the signal name that is used internally in the underlying GTK+ (e.g., `button_press_event`).

Because the universe framework is supposed to offer high-level abstraction of the event handling without exposing the underlying implementation, it is not desirable to show such error messages. The user of the universe library does not know such event names. Possible improvement would be to wrap the call-back functions with a `try ... with` block when they are registered to `big-bang` or `universe` to show which handler raised an exception.

7.4.3 Exception in the communication-related call-back functions

Error messages related to call-back functions become more serious for communication-related events, such as `on_receive`, `on_new`, and `on_msg`. When such communication-related call-back functions raise an exception, the following error message is shown:

```
(process:*process number*):
LablGTK-CRITICAL **: GIOChannel watch: callback raised an exception
```

which does not even show the name of the raised exception. The error message is not the same as the one for other event handlers, because they are handled differently in the universe library: the tick, key, and mouse events are handled as interupts, while communication-related events are processed by watching a communication channel, hence `GIOChannel watch` in the error message.

We could wrap the event handler with a `try ... with` block to show which event was being handled when the exception was raised. In this case, however, we also want to show the name of the exception. Currently, we do not have a good solution to show the name of the exception, other than modifying the underlying LablGTK to show the exception.

7.4.4 Type mismatch between sender and receiver

In the universe library, messages are first marshalled before sent to destination. Due to the specification of the `Marshal` module in the OCaml standard library, the type of the marshalled data is lost during communication. Thus, if a program unmarshalls received data as a value of different type, the program crashes with the unfortunate `Segmentation fault`.

Because the program crashes, we cannot catch the error with a `try ... with` block. Devising a method to statically check the consistency of the type of messages appears to be fundamentally difficult, unless communication patterns are restricted to simple cases. We do not have any good idea to handle this error, other than telling students to check the type of messages whenever a program crashes with `Segmentation fault`.

7.4.5 Communication-related errors

If a server stops while clients are still running, clients continue to run except that they no longer receive any messages from the server. When they send messages to the server, no error arises but the sent messages are simply ignored. This is due to specification of `flush` used to flush the sent messages: it simply sends messages through a channel without checking if the receiver is still running. Although no error arises in this case, from the educational point of view, it would be better to show an error message saying that the receiver is not running.

If we launch a client before a server, we get the following error message:

```
Fatal error: exception Failure("No available connection")
```

We hope that this error message is understandable.

8 Related work

A key to incorporating interactive game programming into education is the simple algebraic interface to world and universe framework. Without being disturbed by I/O operations, game programming can be used to teach mathematics with fun in K-12 education [4]. The idea is incorporated into the second edition of *How to Design Programs* (HtDP2e) [5], and is widely used. Ramsey [11] reports experience of using HtDP2e in an undergraduate course and reorganizes the original six-step design recipe [3] into eight steps, that matches well to the world programming in the introductory course. Morazan [10] reports that students who finished HtDP can deepen their understanding of programming through distributed programming offered by the universe framework and presents a design recipe for distributed programming.

Functional programming and games are linked from different perspective by Matsushima, Ueno, Morihata, and Ohori [9]. They discuss that structured programming thourgh higher-order functions and program transformation based on clear language semantics are suitable for specifying strategies common to game programming as well as logic of games. They specify rules of a game declaratively using a functional language and show how to transform them into a working program.

Bricklayer [13] is an API written in SML to create a 3D LEGO screen, intended as an introduction to the functional programming language SML. With relatively small description, one can obtain various 3D images.

9 Conclusion

In this paper, we reported the cross validation of the universe framework in OCaml by first porting the universe teachpack in Racket to OCaml and then using the resulting universe library in a class room. Although there is room for improvement (especially in the error messages), the universe library in OCaml gives us a nice environment for students to write interactive games, as the universe teachpack in Racket.

In the user testing, most of the students who took the course had already finished the CS1 course. We have not incorporated the universe library into the CS1 course yet (as was done in HtDP2e). Doing so is left as future work.

References

[1] Kenichi Asai (2005): *OCaml Course in Ochanomizu University*. *Presented at "Tips and Tricks" session of the ACM SIGPLAN Workshop on Functional and Declarative Programming in Education (FDPE '05)*. `http://pllab.is.ocha.ac.jp/~asai/papers/fdpe05.html`.

[2] John Clements, Matthew Flatt & Matthias Felleisen (2001): *Modeling an Algebraic Stepper*. *Programming Languages and Systems, Lecture Notes in Computer Science* 2028, pp. 320–334.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2001): *How to Design Programs, An Introduction to Programming and Computing*. The MIT Press, Cambridge, Massachusetts, London, England.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2009): *A Functional I/O System, or, Fun for Freshman Kids*. *ACM International Conference on Functional Programming (ICFP '09)*, pp. 47–58.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2014): *How to Design Programs, Second Edition*. `http://www.ccs.neu.edu/home/matthias/HtDP2e/`.

[6] Matthias Felleisen, David Van Horn, Conrad Barskiand, M.D. & Eight Students of Northeastern University (2013): *Realm of Racket, Learn to Program, One Game at a Time!* No Starch Press.

[7] Robert Bruce Findler & Matthias Felleisen (2002): *Contracts for higher-order functions*. *ACM International Conference of Functional Programming (ICFP '02)*, pp. 48–59.

[8] Yuki Ishii & Kenichi Asai (2014): *Report on a User Test and Extension of a Type Debugger for Novice Programmers*. *3rd International Workshop on Trends in Functional Programming in Education, Electronic Proceedings in Theoretical Computer Science* 170, pp. 1–18.

[9] Yusuke Matsushima, Katsuhiro Ueno, Akimasa Morihata & Atsushi Ohori (2011): *Derivation of game programs from declarative specification in functional language (in Japanese)*. *The 13th JSSST Workshop on Programming and Programming Language (PPL 2011)*, pp. 193–207.

[10] Marco T. Morazan (2013): *Functional Video Games in CS1 III, Distributed Programming for Beginners*. *Trends in Functional Programming, Lecture Notes in Computer Science* 8322, pp. 149–167.

[11] Norman Ramsey (2014): *On Teaching How to Design Programs, Observations from a Newcomer*. *ACM International Conference on Functional Programming (ICFP '14)*, pp. 153–166.

[12] Kanae Tsushima (2013): *Practicable type debugging for functional languages*. Ph.D. thesis, Ochanomizu University, `http://www.ccs.neu.edu/home/matthias/HtDP2e/`.

[13] Victor Winter (2014): *Bricklayer: An authentic introduction to the FPL SML*. *The 3rd International Workshop on Trends in Functional Programming in Education (TFPIE 2014)*.