LablGtk2を用いたuniverse teachpackの実装

上原 千裕　浅井 健一

この研究では，LablGtk2 を用いて Racket の universe teachpack を実装する．universe teachpack には，簡単にゲームを作るための基本的な関数が一通り定義されている．Racket は静的型付けの言語でなく，動的な型付けを行う．そこで，これと同じ働きをして，静的な型付けを行うものを OCaml で実装しようと考えた．今回は，画像の表示やイベント処理に優れている GTK+へのインターフェースである，LablGtk2 を使用した．LablGtk2 を使ったことによって生じる universe teachpack との違いは多少はある．しかし，本質的な動きは universe teachpack と同じであり，静的な型付けが行われる状態でゲームを簡単に作ることができる環境を実装できた．

We implement universe teachpack of Racket using LablGtk2 in this research. The basic functions to make games easily are defined in the universe teachpack. We implement the same library as the universe teachpack of Racket in the statically-typed language OCaml. In this research, we use LablGtk2 that is an Objective Caml interface to GTK+ because that is superior to show images and event processing. There are some differences between universe teachpack of Racket and the universe library we made and they are caused by using LablGtk2. However, the essential behavior of the universe library is the same as that of universe teachpack of Racket. We could implement the environment to make games easily with static typing.

1 はじめに

Racket の universe teachpack [2] を使うと，簡単にゲームを作ることができる．universe teachpack には，ゲームを作るときに必要となる基本的な関数が全て定義されているため，ゲームの作製者はゲームの内容を作ることだけに専念できるからである．universe teachpack を OCaml で実装したのが今回の研究である（以後，今回実装したライブラリを OCaml の universe library と呼ぶ）．静的型付けの言語で実装することで，ゲームプログラミングにおいて，コンパイル時に型が合っているかどうかの確認を行うことを可能にした．これにより，実行前にプログラムの間違いを見つけられる可能性が大幅に増えた．

本稿では，まず第 2 節で Racket の universe teachpack の説明をする．それを踏まえて，第 3 節では LablGtk2 を用いて universe teachpack と同じ働きをする OCaml の universe library を実装をした手順を説明する．第 4 節では，一人で遊ぶゲームと複数人で通信して遊ぶゲームを実装し，我々が作った OCaml の universe library によりゲームのプログラムが実際にどう書けるかを示す．第 5 節で関連研究について述べ、第 6 節でまとめと今後の課題を述べる．

Implementation of universe teachpack using LablGtk2
Chihiro Uehara,Kenichi Asai, お茶の水女子大学理学部情報学科, Dept. of Information and Computer Science, Ochanomizu University.

2 universe teachpack の説明

2.1 ゲームを作るための Racket の関数や型

Racket の universe teachpack にはゲームを作るための基本的な関数や型が定義されている．今回の研究では，それらを LablGtk2 を用いて実現するので，まずはそれらを理解することから始める．

説明のために，白いウィンドウの上を，`image1` という画像が 0.3 秒ごとに x 軸方向に 10 進んでいくゲーム画面（これを game1 とする）を universe teachpack で実装することを考える．すると，

1. 画像の型
2. 画像を重ねる関数
3. 現在の状態に基づいた画面を描画する関数
4. 一定時間ごとに現在の状態を変化させる関数
5. 3.，4. の関数を受け取ったら，その通りにゲームを進める関数

が必要になる。image1 という画像が前の位置から 10 進んだ画像に 0.3 秒ごとに切り替えることにより，動いているように見せる．

1. ～ 5. の説明に入る前に，「世界の情報」である world について説明する．「世界の情報」とは「現在のゲームの状況」のことである．game1 では，image1 の座標を常に把握しておく必要があり，逆に言えばそれさえ把握していれば今作りたいゲーム画面は書ける．よって，game1 の world は image1 の座標になる．この world を変化させることによってゲーム画面が変化し，world を見てゲーム画面が描かれる．game1 だと，0.3 秒ごとに，画面を表示する前に world の x 座標のみ 10 増やして，更新された world をもとにゲーム画面を作り表示する．

2.1.1 画像の型

画像の型には，image 型と scene 型がある．

Racket では，数字などの他に，画像にも名前を与えることができ，

```
(define image1 ***)
```

と書く．***の部分に画像をペーストすると，以後 image1 という名前でこの画像を指すようになり，image1 は image 型になる．

Racket では，画像（image）を重ねて scene と言われる風景を作る．つまり，image 型を重ねて作られるのが scene 型である．

2.1.2 画像を重ねる関数

scene を作るために，universe teachpack に定義されている以下の二つの関数を使う．

```
empty-scene : num * num -> scene
```

この関数には，引数として scene の縦と横の長さを受け取り，その大きさの真っ白い長方形の scene を返す．

```
place-images :
image list * posn list * scene -> scene
```

この関数には，引数として配置したい画像のリスト，画像を置く場所の座標のリスト，画像が置かれる scene を受け取る．3 番目に受け取った scene の，2 番目の引数である座標のリストの一番前の座標に，1 番目の引数である画像のリストの一番前の画像を配置する．これを繰り返すことによって画像のリストに入っている全ての画像を重ねて，全て重ねてできた scene を返す．

2.1.3 現在の状態に基づいた画面を描画する関数

この関数はもともと用意されている関数ではなく，作りたいゲームに合わせて empty-scene や place-images を使って自分で作る．

通常 draw 関数と名付けるこの関数は，現在の状態，つまり world を受け取ったらそれに基づいた画像を返す．

```
draw : world -> scene
```

game1 だと，world（image1 の座標）を受け取ってそこに image1 を出すので以下のようになる．

```
(define (draw world)
  (place-images (list player)
                (list world) background))
```

また，player と background は，

```
(define width 300) ; ウィンドウの横の大きさ
(define height 200) ; ウィンドウの縦の大きさ
(define player "image1.jpg") ; 画像を読み込む
(define background
  (empty-scene width height)) ; 白い長方形
```

などと定義しておく必要がある．

2.1.4 一定時間ごとに状態を変化させる関数

この関数ももともと用意されている関数ではなく，自分で作る．

今，仮に move-x-on-tick 関数と名付ける．この関数は呼ばれるたびに，world（game1 だと image1 の座標）を受け取って更新して返す．具体的には，座標を受け取ったら x 座標のみ 10 足した座標を返すため，以下のように書ける．なお，posn-x と posn-y は Racket に定義されている関数で，座標を受け取ったらそれぞれ x 座標の値，y 座標の値を返す．また make-posn も Racket に定義されている関数であり，引数を 2 つ受け取ってそれらを座標にする．

```
(define (move-x-on-tick world)
  (make-posn (+ (posn-x world) 10)
             (posn-y world)))
```

2.1.5 ゲームを進める関数

これは big-bang 関数と言い，universe teachpack に定義されている関数である．

`big-bang : world clause ...`

この関数は，一つ目の引数として world の初期値を受け取り，二つ目以降の引数としてゲームの動作に必要な関数などを受け取る．game1 では，

1. 第 2.1.3 節の関数 draw
2. 第 2.1.4 節の関数 move-x-on-tick

を big-bang に登録することが必要である．実際に 1. と 2. を big-bang に登録すると以下のようになる．

```
(big-bang (make-posn 20 0)
          (to-draw draw width height)
          (on-tick move-x-on-tick 0.3))
```

on-tick に登録された move-x-on-tick 関数は 0.3 秒ごとに呼ばれる．world が更新されると，to-draw に登録された draw 関数に world が渡されて，draw 関数が返した picture がゲーム画面として表示される．その結果，image1 が動いて見えるゲーム画面になる．

big-bang 関数には，他にも登録できることがたくさんあり，例えばマウスイベントが起きるごとに world を更新する関数や，world の更新を停止する条件（つまりゲーム終了の条件）を設定する関数などが登録できる．

2.2 通信する環境を作る方法

2.2.1 universe teachpack の通信の概要

第 2.1 節で，ゲーム画面を作る説明をしたが，作れるようになったのは 1 人で遊ぶゲームだけである．複数人で対戦するためには，サーバとクライアントを作ることが必要である．

universe teachpack では，プレイヤーのプログラム，すなわち big-bang 関数を使って書いたプログラムは全てクライアントである．クライアントとは別にサーバを 1 つ作り，クライアントとサーバが情報を送受信することで通信を行う．このやり取りする情報に型の制限はない．以後，このやり取りする情報のことをメッセージと呼ぶ．

例えば，2 人（クライアント 1 とクライアント 2 とする）で game1 をそれぞれ動かし，相手の image1 の x 座標を自分のゲーム画面の左上に表示することを考える．これは，お互いが相手の image1 の x 座標を知ることができれば実現できる．クライアント 2 がクライアント 1 の x 座標を得る手順を考えると

1. クライアント 1 が自分の image1 の x 座標（x 座標 1 とする）をメッセージとしてサーバに送る
2. サーバはクライアント 2 に x 座標 1 をメッセージとして送る
3. クライアント 2 はクライアント 1 の x 座標をサーバからのメッセージとして得る．

となる．逆のクライアント 1 がクライアント 2 の x 座標を得る手順も同じ考え方である．先ほど述べたように，メッセージは型が決められていないので，数字以外でももちろん送ることができる．そのため，メッセージとして x 座標を送るのではなく，自分の x 座標が入った world をサーバに送り，相手はサーバからメッセージを受け取ってそのメッセージの中から x 座標を取り出しても同じことができる．

big-bang 関数には今まで，world を返す関数を登録していたが，通信をする時はその代わりに「world か world とメッセージの組」(これを universe teachpack では Package という）を返す関数を big-bang 関数に登録する．もし登録された関数が world だけを返したら，メッセージは送られず world が更新されるだけであり，world とメッセージの組を返したら，world が更新されてメッセージがサーバに送られる．

また，big-bang 関数の on-receive には，「メッセージを受け取るごとに呼ばれて Package 型の値を返す」という関数が登録できる．これを登録すると，サーバからメッセージが送られて来るたびにこの関数が呼び出されるので，world とメッセージの組を返す関数を登録すればメッセージがサーバに送られる．

big-bang 関数に必要な関数を登録することによって，ユーザのプログラムがクライアントになることは分かった．後はサーバの作り方が必要である．

2.2.2 サーバを作る関数 universe

universe teachpack には `universe` という関数があり，これがサーバを作るときに使う関数である．クライアントのプログラムを作る時に使う `big-bang` と同じように，関数などを `universe` に登録することでサーバとしての役割を果たし，通信を可能にする．

```
universe : state-expr clause ...
```

`state-expr` は，`big-bang` の `world` と同じような役割を果たす．すなわち，サーバの現在の情報を持っていて，`universe` に登録された関数が呼ばれるたびに `state-expr` が変更される．`world` と同様に，`state-expr` の型は何でもよい．簡単なサーバの例を挙げる．

```
(universe init-state
          (on-new new-expr)
          (on-msg msg-expr))
```

通信に新たなクライアントが参加したら，`new-expr` 関数が呼ばれ，クライアントからメッセージを受信したら `msg-expr` が呼ばれる．

2.2.3 通信をするゲームを作る時にやるべきこと

通信ゲームを作るときは，実装を始める前に，クライアントとサーバの通信がどのように成り立っているかを図にすることが大切である．図を書いてから実装しないと，メッセージのやり取りがどのように起こっているのか分からなくなり，いざプログラムを実行した時に思い通りに通信ができないことがある．

例えば，図 1 は，二人対戦のゲームの通信が始まる時の様子の一例である．二人のプレイヤーが通信に参加したことをサーバが把握したら，二人のプレイヤーにそれぞれ「start」というメッセージを送っている．

3 OCaml の universe library の実装

Racket の universe teachpack が一通り分かったところで，それを LablGtk2 を用いて実装する．これを OCaml の universe library と呼んでいる．これに実装すべきものは大きく分けて 2 つあり，

1. クライアントのプログラム（`big_bang` 関数を使ったゲームの画面を作るプログラム）を作るための関数と型
2. サーバのプログラム（`universe` 関数を使った通信の時のみ使われるプログラム）を作るための関数

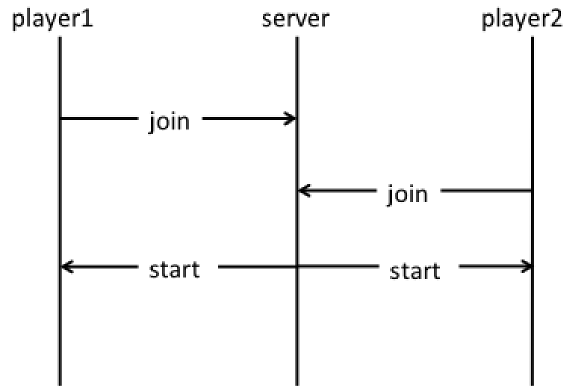

図 1 通信が始まる時の様子の例

である．順に説明する．

3.1 クライアントのプログラム作成用の関数と型

クライアントのプログラムを作るためには，

1. 画像の型と画像を読み込む関数（`image` 型）
2. 重なった画像の型と画像を重ねる関数（`scene` 型，`place-images` 関数）
3. ゲーム画面を作る関数（`big-bang` 関数）
4. `world` か，`world` とメッセージの組を表す型 (Package 型)

が必要とされる．それぞれ，Racket の universe teachpack だと何に対応するかを横に書いた．universe teachpack では，画像は直接プログラムに貼っていたので，画像を読み込む関数に対応する関数はない．以下 1. ～ 4. を説明した後，3. の実装について説明する．

3.1.1 画像の型と画像を読み込む関数

universe teachpack では，`image` 型という画像を表す型があり，プログラムに直接画像を張ってその画像を `image` 型として読み込んでいた．OCaml の標準ライブラリだけでは，画像を柔軟に扱うことができない．後々，マウスのクリックイベントなどを扱いたいことも考慮して，画像の表示やイベント処理に優れている GTK+へのインターフェースである，LablGtk2

を使用することにした．

Lablgtk2 でアンチエイリアスをかけて表示するためには，表示する時ごとに画面を全て書き直さなくてはいけない．そのため，表示する時に必要となる情報を持った画像の型である gnopixbuf 型を定義した．

```
type gnopixbuf = {
 pixbuf : GdkPixbuf.pixbuf;(*画像のデータ*)
 x : int; y : int;(*画像を表示する座標*)
 width : int; height : int;(*画像の大きさ*)
}
```

他にも，円や文字などの，Racket では image 型として扱われているものはそれぞれ型を定義した．以下は円とテキストの型である．

```
type gnocircle = {
 color : string;(*円の色*)
 x1 : int; y1 : int;(*円を表示する座標*)
 x2 : int; y2 : int;(*円の横軸と縦軸の長さ*)
}
type gnotext = {
 color : string;(*文字の色*)
 text : string;(*文字*)
 x : int; y : int;(*文字を表示する座標*)
 size : int;(*文字の大きさ*)
}
```

そして，これらの型をまとめて一つの型に入れることにより，見かけ上は universe teachpack の image 型と同じ，画像を示す型ができた．この型に picture と名付けた．

```
type picture =
  RECT of gnorect (*長方形*)
 |CIRCLE of gnocircle (*円*)
 |TEXT of gnotext (*文字*)
 |PIXBUF of gnopixbuf (*取り込んだ画像*)
 |IMAGE of picture list (*画像のリスト*)
```

なお，IMAGE of picture list については第 3.1.2 節で説明する．

あとは，画像を読み込んだときに picture 型を返す関数を作ればよい．LablGtk2 では，画像をプログラムに直接貼ることはできないため，画像の名前を書いて読み込むようにした．

```
(*read_image : string -> picture*)
let read_image name =
  let buf = GdkPixbuf.from_file name in
  (*ファイル名から画像データを読み込む*)
    (PIXBUF {pixbuf = buf ;x = 0;y = 0;
    (*とりあえず座標は原点にしておく*)
    width = GdkPixbuf.get_width buf;
    (*画像の横軸の長さをとる*)
    height=GdkPixbuf.get_height buf
    (*画像の縦軸の長さをとる*)})
```

画像を表す型が書けたので，第 2.1.2 節で示した empty_scene 関数などの画像を作る関数は作れるようになった．

3.1.2 重なった画像の型と画像を重ねる関数

universe teachpack では，image 型の値を重ねて scene 型の値を作っていたが，画像データであることは変わりなく，型を分ける意味がないように思えたので，Racket の image 型と scene 型は picture 型に統一した．画像のリストと座標のリストと一番下の画像をもらってきたら全てを重ねた画像を作る関数を作った．

```
place_images :
picture list -> (int * int) list -> picture
                                  -> picture
```

これは，再帰関数として実装する．第一引数である画像リストのそれぞれの座標を，第二引数である座標リストを使って変更し，座標を変更した画像リストの頭にコンストラクタとして第 3.1.1 節の IMAGE をつけることによって，表示したい画像のリストが値である picture 型を返している．画像のリストにしておくことによって，draw 関数にそれが渡された時に画像を順番に重ねることができ，ユーザからは place_images 関数によって重なった画像が作られていたように見える．

3.1.3 ゲーム画面を作る関数

world の初期値以外は，オプション引数として実装することで，universe teachpack と同じように使いたい関数のみ登録すること，また関数の順番を気にせずに登録することができる．

big_bang 関数の動き方は以下である．

1. 登録された draw 関数を使って，初期 world を描写する
2. 一定時間ごとに world を変化させる関数を呼んで，world を更新する
3. キーイベントやマウスイベントが起きたときも，対応する関数を呼んで world を更新する
4. world が更新されるたびに，新しい world を元に draw 関数を使って描写する

このように動くと，登録した関数が呼ばれる条件が満たされるたびに world が変更され，world が変更されるたびにゲーム画面が変更されるという big_bang の目的が達成されている．なお，1. と 4. の draw 関数は，第 2.1.3 節で説明した draw 関数と同じ働きをする．draw 関数はユーザが書くものであり，クライアントのプログラムを作るための関数ではないので，ここではなく後に実装の方法を説明する．

また，第 2.2.1 節で示したように，ゲーム画面を作る関数 big_bang は，サーバとメッセージを送受信できるように実装しなければならない．OCaml で通信を行うために，今回は Unix モジュールを使ってサーバ・クライアント通信ができるように実装した．

3.1.4 world か，world とメッセージの組の型

通信をする時のために，第 2.2.1 節で述べた Racket の universe teachpack の Package 型に対応する，「world か world とメッセージの組」を表す型を作る必要がある．world の具体的な型はユーザしか知り得ないため，world の型は決められない．メッセージもまた同じである．よって，world の型を'a，メッセージの型を'b とする多相型として定義する．

```
type ('a,'b) handleresult =
  World of 'a
 |Package of 'a * 'b
```

このように定義することで，world のみの場合は World world，world とメッセージの組の場合は Package (world,message) と書くことができ，型が定義できた．ゲーム画面を作る関数 big_bang は，登録された handleresult 型を返す関数が World world という値を返したら world を変更して画面を描き，Package (world,message) という値を返したらそれに加えてサーバに message を送信するように実装する．

3.1.5 big_bang 関数の実装

第 3.1.3 節と第 3.1.4 節を踏まえて，big_bang 関数の実装の主な流れを考える．big_bang 関数は最初に world の初期値を受け取る．仮に firstworld という名前で受け取ったとする．big_bang 関数内の何時でもどこでもただ 1 つの world を使うため，big_bang 関数は最初に ref 関数を使って world を定義する．

```
let world = ref firstworld
```

次に，各イベントに対する処理を登録していく．例として，一定時間ごとに world を書き換える関数を考える．仮にこの関数を on_tick という名前で受け取ったとする．on_tick 関数を，タイマーイベントが発生するたびに呼び，world を更新してそれに合わせて画面を更新し，必要ならばメッセージをサーバに送る．なお，タイマーイベントの発生する時間の間隔も big_bang 関数の引数として受け取っている．

```
(*get_result : handleresult -> unit*)
let get_result result =
  match result with World w ->
    (world := w; (*world の更新*)
     draw_window ()) (*画面を描く*)
   |Package (w,m) ->
    (world := w; (*world の更新*)
     draw_window (); (*画面を描く*)
     send m) (*メッセージを送る*)
```

get_result 関数はユーザの返答を処理する関数である．draw_window 関数と send 関数は，画面を描く関数とメッセージを送る関数であり，別途定義した．

```
let time_event _ =
 (let result = on_tick !world in
 (*更新された world（とメッセージ）を受け取る*)
 get_result result;
 true)
```

登録されている on_tick 関数に world を渡し，ユーザが返した返事を処理する．下のようにタイマーイベントの登録をする時に，コールバック関数は bool 値を返すようになっており，true を返すとこのコールバックの処理後，再びタイマーイベントを起こすようになる．

```
GMain.Timeout.add ~ms:rate
                  ~callback:time_event
(*タイマーイベントの登録．rate は時間の間隔*)
```
他のイベント処理もこのように登録する．

3.2 サーバのプログラムを作るための関数

サーバのプログラムを作るためには universe 関数だけが必要である．第 3.1 節でも述べたように，OCaml で通信を行うために，今回は Unix モジュールを使ってサーバ・クライアント通信ができるように実装した．サーバの状態の初期値以外は，オプション引数としてもらうように実装し，big_bang 関数と同じように使いたい関数のみ登録できるようにする．

サーバ・クライアント通信をする際に，送受信するメッセージの型の制限をしていないため，クライアントがメッセージを送ろうとした時にそのまま送ろうとすると送れなかったりデータが壊れてしまう場合がある．そのため，今回はメッセージを marshalling して string 型に変換し，その文字列を送受信した．その際，Unix モジュールの read 関数と write 関数を使った．また，big_bang 関数には，サーバから文字列を受け取ったらそれを unmarshalling する処理を書いた．このようにメッセージを marshalling することで，データを確実に送受信することができた．なお，ユーザはゲームを作る際に，メッセージを送る側と受け取る側で同じ型を扱うように設計しなければならない。

3.3 実装の順番

OCaml の universe library の実装は，まず通信に関わらないことを全て実装してから通信に関わることを付け加えていくべきである．前者と後者は独立しており，それらを同時平行で進めようとするとプログラミングをしていてややこしく感じるからである．

具体的には，最初はクライアントのプログラムを作るために必要な関数や型だけを実装し，かつ第 3.1.4 節で述べた型を作らないで world のみを扱うように実装する．そこまで実装できたら，それを使って実際にゲーム画面を作ってみて，思い通りに動くか確かめる．その後，サーバのプログラムを作るために必要な関数を作り，クライアントのプログラムにサーバと通信できる仕組みと第 3.1.4 節で述べた型を加える．

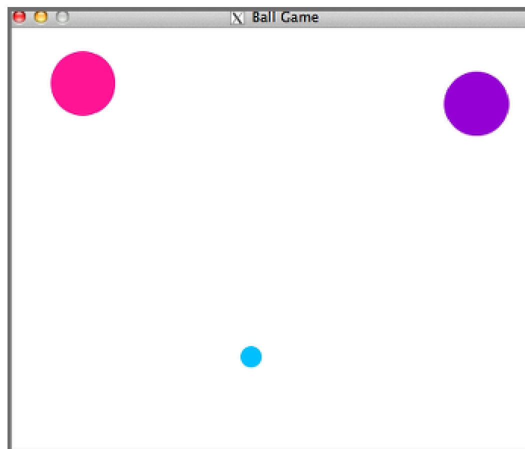

図 2　ボールゲーム

4 ゲームの実装

第 3 節で作った OCaml の universe library を使って，実際にゲームを作る．まずは通信なしの一人で遊ぶゲームを作って，ゲーム画面の表示に関する理解を深め，その次に通信ゲームを作って通信に対する理解を深める．通信ゲームは, 理解しやすいように二人で通信するゲームを作った．通信ゲームを作る時も，第 3.3 節同様，最初に通信に関わらない部分を実装し，いったんゲームが思い通りに動くことを確かめてから通信に関する部分の実装をするべきである．

4.1 1人で遊ぶゲーム

ゲーム画面には色の違うボールが 3 個動いている．ボールをクリックするとボールは小さくなり，3 個のボール全てが決められて大きさよりも小さくなったらクリアというルールである（図 2）．

world の定義と big_bang 関数の呼び出しは，以下になる．

```
type ball_t = {
  center : int * int; (*中心の座標*)
  vector : int * int; (*方向ベクトル*)
  radius : int; (*半径*)
  color : color_t; (*色*)
}
```

```
type world = {
  lob : ball_t list;
  (*ボールの情報のリスト*)
}
big_bang initial_world
         ~name:"Ball Game" (*ウィンドウ名*)
         ~to_draw:draw
         ~x:width (*ウィンドウの横の大きさ*)
         ~y:height (*ウィンドウの縦の大きさ*)
         ~on_tick:change_world_on_tick
         ~rate:0.3
         ~on_mouse:change_world_on_mouse
         ~stop_when:game_over
         ~last_scene:draw_game_over
```

change_world_on_tick 関数は，登録された rate 時間ごとに呼ばれ，その都度 world を更新する．change_world_on_mouse 関数は，カーソルがウィンドウ内にある状態でマウスが押されるか離されるかしたら呼ばれ，world を更新する．game_over 関数は，world をもらってきたら真偽値を返す．もしこれが true を返せば，もうゲームは終了して画面を更新しない．draw_game_over 関数は，ゲームが終了したらこの関数が呼ばれ，world が渡されて picture を返す．つまりゲームの終了画面を作る．

draw 関数は，world を受け取ったら，それに合わせて place_images 関数を使って画像を重ねて新しい画像を作るように実装する．以下のように定義した．

```
let draw {lob = lst} =
 (place_images (List.map ball_image lst)
               (List.map ball_center lst)
                              background)
```

ここで，ball_image 関数は，world から描きたいボールの画像のリストを作り，ball_center 関数は，ボールの中心の座標のリストを作る．

また，change_world_on_tick 関数は以下のように定義した．

```
let change_world_on_tick {lob = lst} =
 World {lob = List.map move_ball_on_tick lst}
```

ここで，move_ball_on_tick 関数は，ボールの方向ベクトルに応じてボールの座標を変更する関数である．

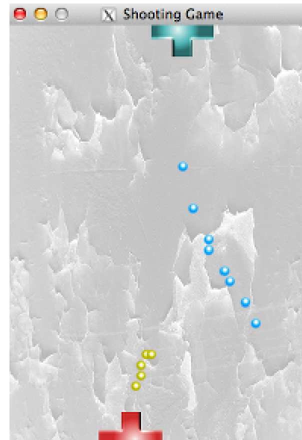

図 3　シューティングゲーム

4.2　複数人で通信して遊ぶゲーム

相手と向かい合って，相手に弾を当てたら勝ちになるシューティングゲームを作る（図 3）．左右の矢印キーで自分を左右に動かし，上の矢印キーで弾を発射する．

通信ゲームなので，まずは，第 3.3 節で述べたように，通信をしないときにクライアントの画面がどうなるかを考えてから通信に関する実装をする．通信をしない場合，クライアントができることは，左右の矢印キーで自分のキャラクターを左右に動かすことと，上の矢印キーで弾を発射することのみである．よって，クライアントの world の定義は以下になる．

```
type world = {
 posn : int * int; (*自分の座標*)
 ball_posn_lst : (int * int) list;
      (*自分が打った弾の座標のリスト*)
}
```

あとは，キーのクリックを受け取る関数と一定時間ごとに ball_posn_lst のすべての座標を増やす関数を big_bang 関数に登録すれば，通信をしない場合のクライアントのゲーム画面は完成である．

通信をしない場合についての実装ができたので，次は通信に関する実装をする．具体的には，サーバを作ることと，通信をすることに伴うクライアントのプログラムの変更が必要になる．まず，通信がどのように行われているかの図を考える．二人のクライアントが

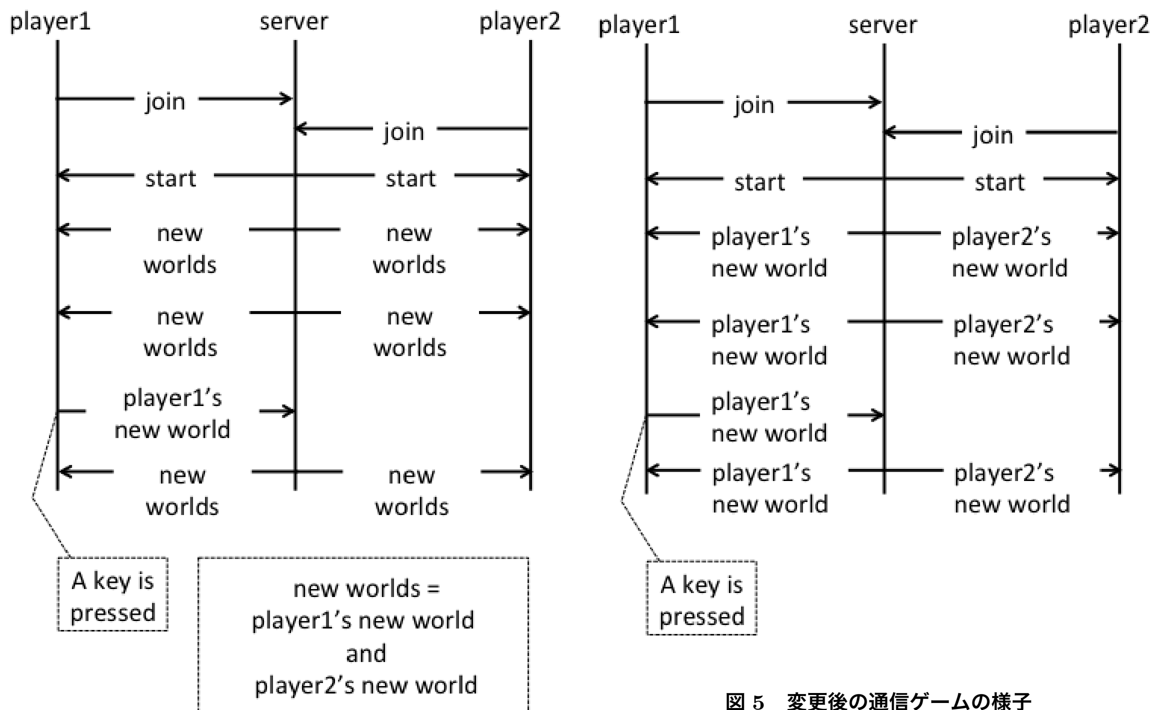

図 4　通信ゲームの様子

図 5　変更後の通信ゲームの様子

同じ時間軸で動くために，状況の変更は全てサーバが行う [4]．また，自分の画面に相手のキャラクターや相手が打った弾を表示させるため，クライアントは対戦相手の最新の状況も自分の最新の状況とともに常に持っておく必要がある．よって，通信の様子は図 4 のように書ける．

では，具体的な通信に関する実装に移る．まずクライアントのプログラムの変更について考える．通信の図を見て分かる通り，キーが押されたら自分の world をサーバに送り，サーバからは常にクライアント二人分の world を受け取りたい．ここで，第 3.2 節で述べた通り，メッセージを送る側と受け取る側で同じ型を扱うようにしなければならない．送るものと受け取るものの型をそろえるため，クライアントの world が自分の状況も相手の状況も含むものに変更する．

```
type world = {
 posn : int * int; (*自分の座標*)
 ball_posn_lst : (int * int) list;
      (*自分が打った弾の座標のリスト*)
 rival_posn : int * int; (*相手の座標*)
 rivalball_posn_lst : (int * int) list;
      (*相手が打った弾の座標のリスト*)
}
```

これで，サーバに送るものも受け取るものも，自分の world だけでよくなった．通信の様子は図 5 のように書けるようになる．player1 と player2 ではどちらが「自分」でどちらが「rival」なのかが異なるだけなので，player1’s world と player2’s world では，「自分」の情報と「rival」の情報が入れ替わったものになっている． また，big_bang 関数への登録も少し変更し，以下のようになった．

```
big_bang initial_world
         ~name:"Shooting Game"
         ~to_draw:draw
         ~x:width
         ~y:height
         ~on_key_press:key_draw
         ~rate:0.3
         ~register:client_sockfd
         ~on_receive:getmessage
```

```
          ~stop_when:game_over
          ~last_scene:draw_game_over
```

サーバに on_tick 関数を任せるため，クライアントの big_bang 関数への登録から on_tick 関数をなくした．また，クライアントは，キーが押された時のみサーバにメッセージを送りたい．よって，key_draw 関数は Package コンストラクタがついた world とメッセージの組を返す．getmessage 関数は，更新された world をもらうので，

```
let getmessage world mail = World mail
```

となる．これで world が最新のものに更新される．また，client_sockfd は，サーバ・クライアント通信に必要な，既に起動したサーバプログラムのポート番号と IP アドレスの組である．

次にサーバを考える．サーバがクライアントの world の変更を行うため，state の中に各クライアントに送った world，つまり各クライアントの最新の world を持っている．よってサーバの state はクライアントの名前と world の組のリストになる．サーバは，新しいクライアントが加わったときに呼ばれる関数 onnew を，

```
let onnew state iworld = ...
```

と定義しており，iworld という引数がクライアントの区別をするもの，つまり名前になっている．よって今回の場合なら state は，

```
{[(player1's iworld, player1's world);
  (player2's iworld, player2's world)]}
```

となる．universe 関数の呼び出しは以下になる．

```
universe init_state
          ~on_new:onnew
          ~on_msg:onmsg
          ~rate:0.3 (*tick関数が呼ばれる間隔*)
          ~on_tick:tick (*画面の弾を進める*)
```

onmsg 関数は，クライアントからメッセージが来たときに呼ばれる関数である．クライアントからメッセージが来るのはキーが押された時だけである．メッセージを送ってきたクライアントにはメッセージとして world を返すとともに，メッセージを送って来なかった方のクライアントには，world の rival_posn と rivalball_posn_lst を変更したものを送る．

5 関連研究

5.1 関数型言語とゲーム作成

本稿は、自分の書いたプログラムが間違っていないかチェックしやすいという理由で，静的型付けの言語である関数型言語の OCaml を選択した．ゲームと関数型言語の関係についてさらに追求して考えているのが松島，上野，森畑，大堀 [3] である。彼らは型システムの他に，高階関数を用いたプログラムの構造化，明瞭な意味論に基づいたプログラム変換などが関数型言語では深く議論されていて，これらの技術はゲームプログラムに共通する定石やゲームに関する論理を記述するのに有効である可能性があると述べる．そして彼らは、関数型言語におけるプログラミング手法の内主要なものの 1 つである，宣言的な記述によるプログラミングに注目し，ゲームのルールの宣言的な分析と記述に基づいたゲームプログラミング手法を述べた．彼らは簡単なゲームでの実際の例を見せており，今後は複雑なルールへの応用をしたいと述べていた．彼らの研究から，関数型言語でのゲームプログラミングは便利なプログラミング環境であり，まだまだ発達できることが伺える．

5.2 初心者のゲーム作成

HtDP [1] は学んでいて，ゲームプログラミングに関しては初心者である学生が，Racket の universe teachpack を使ってゲームを作る授業を通して，どのような考え方やプログラムを生み出して理解していったのかが Morazan [4] に述べられている．この論文には，学生が最初に考えついた一見合ってるように見えるが実は避けた方がよい考え方なども書かれており，ゲームプログラミングの初心者がどのようにして理解を深めていくのかが興味深かった．この論文中では初心者が使う基本的な関数のみが出てきてゲームを作っていたため，私はこの研究を本格的に始める前に一度読み，最低限なくてはいけない関数や仕組みを理解した．今後様々な人に今回実装した universe library を使ってゲームを作ってもらおうと考えているが，ゲームプログラミングの初心者に参加してもらう予定であるから，この論文はその時の手助けになると思う．

5.3 関数型言語と画面の表示

ゲームを作る上で，表示される画面がどうなるかはとても重要である．本稿では，LablGtk2 を用いることによって，図形をアンチエイリアスをかけて描けることに加えて，jpg 画像や png 画像などの画像データも扱えるように実装することで，表示する画面をなるべくきれいでユーザの思い通りになるようにした．

画面の表示の関連で，興味深いのが Winter [5] で紹介されている Bricklayer である．これは，SML で書かれた API で，LEGO® 風の 3D 画面を表示できるものである．論文 [5] 中に，Bricklayer は初心者の関数型言語への導入にちょうど良いと書かれている通り，少ない分かりやすい記述で色々な図形の表示ができる．関数型言語で書かれていて，ユーザの入力が比較的少なくてユーザを楽しませる画面を表示できることは本研究の目標の 1 つであるため，Bricklayer は参考になる．

6 まとめと今後の課題

Racket の universe teachpack とまったく同じではないものの，本質的な部分はほとんど同じ働きをするものが実装できた．違いの例を述べると，画像をプログラムに直接貼れないことなどである．実際に今回作った OCaml の universe library を用いて作った二つのゲーム（一人で遊ぶものと複数人対戦のもの）はエラーを出すことなく思い通りに動き，universe library は実際に使える段階まで進んだ．

現段階では，`image` 型と `scene` 型を `picture` 型に統一したこと以外は Racket の universe teachpack の動きを真似することしかしていないが，自分たちが使いやすいように改良していきたい．

今後は，universe library を使って様々な人にゲームを作成してもらい，たくさん出てくるであろう問題を解決していきたい．今までで作ったゲームは，この研究の中身を知っている人が試しに作った簡単なゲームしかない．なので，様々な人に使ってもらうことで初めて見える欠点やバグがあると思う．それらを改善して，静的型付けの言語でのゲーム作成の環境をより整えていきたい．

参考文献

[1] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S.: *How to Design Programs: An Introduction to Programming and Computing*, MIT Press, Cambridge, MA, USA, 2001.

[2] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S.: A Functional I/O System or, Fun for Freshman Kids, *ICFP 2009*, (2009), pp. 47–58.

[3] 松島勇介, 上野雄大, 森畑明昌, 大堀淳: 宣言的記述からの関数型言語によるゲームプログラムの導出, 第 *13* 回プログラミングおよびプログラミング言語ワークショップ（*PPL2011*）論文集, (2011), pp. 193–207.

[4] Morazan, M. T.: Functional Video Games in CS1 III, *TFP 2013*, (2013), pp. 149–167.

[5] Winter, V.: Bricklayer: An authentic introduction to the FPL SML, *TFPIE 2014*, (2014).