Yabaitech Tokyo Vol.7

MasWag　binary　zeptometer　gfn

YABAITECH.TOKYO

vol.7

2021

目次

チルノのパーフェクトフォワーディング (完全転送) 教室　MasWag　1

不完全情報ゲームのナッシュ均衡を CFR (Counterfactual Regret Minimization) アルゴリズムで求めよう　ばいなり　10

炒め物のイディオム　zeptometer　56

自作言語 Sesterl でオンライン対局ゲームを実装した話　gfn　66

チルノのパーフェクトフォワーディング (完全転送) 教室

MasWag

1. はじめに

> みんなぁ〜！
>
> C++11、(だいたい) ⑨ 周年だよ！
>
> 値(あたい) みたいな天才目指して、
>
> 頑張っていってねー！

1.1. コメント

今回は C ⑨⑨ ということで、この記事は C++11 のパーフェクトフォワーディングについての記事です。これを読んで何かがわかるのかは良くわからないですけど生温かい視線で見ていただけると幸いです。真面目に理解しようと思ったら別の記事を読んだ方が良いです。

2. rvalue と lvalue

> キラキラ　C++
>
> 輝く　rvalue
>
> 参照　rvalue(右辺値)　もなんとかして入ろう

関数　変数　トップ目指して GO GO!![1]

2.1. 解説

C や C++ には rvalue (右辺値) と lvalue (左辺値) があります。元々は代入の左辺に置くことのできる値が lvalue (左辺値)、右辺にしか置けない値が rvalue (右辺値) という非常にわかりやすい区別でしたし、「参照」といえば左辺値の参照のことでした。

```
int return_int() {
  return 10;
}
void lvalue_reference(void) {
  // C++11 より前は、右辺にある値が rvalue で左辺にある値が lvalue だった (非常に明快で良い)
  int alice_margatroid = return_int();
  // 当たり前だが、lvalue の参照を取ることができる
  //旧作のアリスは Win 版のアリス・マーガトロイドと同一人物 (らしい)
  int& pc98_alice = alice_margatroid;
}
```

ところが C++11 で、rvalue で rvalue 参照 (右辺値参照) という左辺に置くことのできる rvalue が登場しました ... という話を次の節でしていきます。

3. rvalue 参照の例

> バーカ！バーカ！バーカ！バーカ！バーカ！バーカ！)
>
> チルノ「ちょっ ... 違っ ...rvalue じゃないもん！」
>
> (バーカ！バーカ！バーカ！バーカ！バーカ！バーカ！)
>
> チルノ「左辺にある方が lvalue なのよぉ！」

```
void rvalue_reference(void) {
  // C++11 以降では && で rvalue の参照も取ることができる様になった
```

1　この記事は C++11 についての記事です。Go 言語は関係ない

```
  // chirno は左辺にあるけど rvalue の参照なので rvalue
  //   (正確には eXpiring value、xvalue)
  // rvalue 参照は rvalue だけど life time が延長されて、式の外に出ても使うことができる。
  int&& chirno = return_int();
```

(バーカ！バーカ！バーカ！バーカ！バーカ！バーカ！)

チルノ「なにようるさいわね　この rvalue っ！」

(バーカ！バーカ！バーカ！バーカ！)

ピチューン (life time が終わって死んだ音)

```
  // 普通の rvalue (正確には prvalue) は式が終わると life time が終わる
  return_int();
  // ここで chirno の life time が消える
  // 一方 return chirno; の様にすれば rvalue 参照を返して引き続き life time を伸ばすこともできる
}
```

3.1. 解説

C++11 ではこれまで通りの rvalue (prvalue) に加えて rvalue の参照を作ることができる様になりました。rvalue なのに参照とは？という点はとりあえず置いておいて、rvalue の参照を使うことで rvalue を式の外でも使うことができる様になりました。

4. rvalue の参照の存在意義 : 無駄な copy を防ぐ

ここでは以下の Bus クラスを使います。

```
class Bus {
private:
  char * ptr; // C++11 以降であればスマートポインタを使うべき
public:
  Bus() {
```

```
    std::cout << "default constructor" << "\n";

    ptr = new char[3];
    // 本当は何か意味のあることをやる
  }
  // C++03 以前でもできた、コピーコンストラクタ
  Bus(Bus const & bus) {
    std::cout << "copy constructor" << "\n";
    ptr = new char[3];
    // 受け取ったオブジェクトをコピーする
    std::copy(ptr, ptr + 3, bus.ptr);
  }
  // C++11 以降でできる様になった、ムーブコンストラクタ
  // rvalue の参照を受け取った場合、受け取ったオブジェクトを破壊しても良い
  Bus(Bus && bus) {
    std::cout << "move constructor" << "\n";
    // 受け取ったオブジェクト中の ptr をムーブして破壊する
    ptr = bus.ptr;
    bus.ptr = nullptr;
  }
  void print_ptr() {
    std::cout << std::hex << reinterpret_cast<size_t>(ptr) << "\n";
  }
  bool is_null() {
    return ptr == nullptr;
  }
  ~Bus() {
    delete [] ptr;
  }
};
```

- 紅魔館からバスが出て始めに 3 人乗りました

```
void copy_and_move() {
  Bus koumakan = Bus(); // 内部で 3byte 分のメモリが new される
  koumakan.print_ptr(); // -> 7f8aa9405a80
```

白玉楼にコピーして半人 (はんじん) だけ乗りました

```
// コピーコンストラクタ。static_cast は不要
Bus hakugyokurou = Bus(static_cast<const Bus&>(koumakan));
koumakan.print_ptr();     // -> 7f8aa9405a80
hakugyokurou.print_ptr(); // -> 7f8aa9405a90
```

ここで koumakan と hakugyokurou のアドレスが異なることに注意してください。

八雲さん家にムーブして 結局紅魔の乗客何人だ？

```
// ムーブコンストラクタ。これ以降で koumakan を使ってはいけない
Bus yakumo = Bus(static_cast<Bus&&>(koumakan));
```

答えは答えは　0 人　0 人

なぜならなぜなら　そ・れ・は

メモリ操作でムーブエンド

```
// koumakan.ptr は nullptr なので 0 が出力される
koumakan.print_ptr();     // -> 0
hakugyokurou.print_ptr(); // -> 7f8aa9405a90
// 元々 koumakan.ptr だったアドレスは yakumo.ptr に移った
yakumo.print_ptr();       // -> 7f8aa9405a80
}
```

4.1. 解説

rvalue の参照ですが、 無駄なメモリ上の copy を防ぐために使います。 より正確に言うと、 rvalue の参照として渡されたオブジェクトは破壊してもよい、 というおやくそくがあります。 int や double などのプリミティブ型の場合は「破壊」と言っても特に何も破壊するものもありませんが、 例えば上記例の Bus の場合には与えられたオブジェクトを破壊することで、 コピーコンストラクタにある様にメモリの中身をコピーする代わりに、 ムーブコンストラクタに

ある様にポインタの取り替えだけでオブジェクトを生成することができます。 より具体的には、前半では koumakan をコピーして hakugyokurou を作ることで、7f8aa9405a80 にあるオブジェクトが 7f8aa9405a90 にコピーされました。一方後半では koumakan をムーブして yakumo を作るので、yakumo 中のアドレスが 7f8aa9405a80 となり、一方 koumakan 中のアドレスは 0 になりました。

5. std::move

> キャストに　意味など無いわ
>
> rvalue 立てば　いいのよ
>
> std::move があれば　な・ん・で・も１２⑨！

```
void copy_and_move2() {
  Bus koumakan = Bus(); // 3byte のメモリが確保される
  koumakan.print_ptr();

  // コピーコンストラクタ。static_cast は不要
  Bus hakugyokurou = Bus(koumakan);
  koumakan.print_ptr();
  hakugyokurou.print_ptr();

  // ムーブコンストラクタ。これ以降で koumakan を使ってはいけない
  Bus yakumo = Bus(std::move(koumakan));
  koumakan.print_ptr(); // koumakan.ptr は nullptr なので 0 が出力される
  hakugyokurou.print_ptr();
  yakumo.print_ptr(); // 元々 koumakan.ptr だったアドレスは yakumo.ptr に移った
}
```

5.1. 解説

前述の例では `static_cast<Bus&&>(koumakan)` と書くことで koumakan の rvalue 参照を得ていましたが、これを毎回書くのはなかなか長いです。そこで出てくるのが標準ライブラリの `std::move` です。結局やっていることは `static_cast` と同じですが、`std::move` を使うことで前述の例を以下の様に短く書くことができます。

6. Universal 参照 : みんな大好きテンプレート

ここからは rvalue 参照や lvalue 参照とテンプレートの関係について見ていきます。

> くるくる　template　ぐるぐる　頭回る
>
> だって　&(つぶら目玉)　二つ (まで) しかないのに
>
> 三本の&(針)なんて　ちんぷんかん

```
// T 自体は lvalue 参照か  参照ではない型になる。今回の例では T は int& か int のどちらかになる
template<typename T>
void print_if_ref(T&&) {
  // この static_assert の中身は C++17 が必要
  // T 自体は参照ならばつねに lvalue の参照になる
  static_assert(std::disjunction<std::negation<std::is_reference<T>>,
                                 std::is_lvalue_reference<T>>::value,
                "T is not an rvalue reference.");

  if (std::is_lvalue_reference<T&&>::value) {
    // T が int& の場合、T&& は int& になる (T&&& -> T& と私は覚えた)
    std::cout << "lvalue ref" << std::endl;
  } else if(std::is_rvalue_reference<T&&>::value) {
    // T が int の場合、T&& は int&& になる
    std::cout << "rvalue ref" << std::endl;
  }
}

void universal() {
  int chirno = 9;
  print_if_ref(chirno); // -> lvalue ref
  print_if_ref(9); // -> rvalue ref
  int& chirno_ref = chirno;
  print_if_ref(chirno_ref); // -> lvalue ref
}
```

6.1. 解説

みんな大好きテンプレート、rvalue 参照とはどう組み合わせれば良いでしょうか？より具体的には、lvalue を渡したら lvalue の参照として、rvalue を渡したら rvalue の参照として扱える様なテンプレート関数を書きたいです。

ここで出てくるのが Universal 参照です。Universal 参照ではテンプレート引数 T に対して T&& と書くことで、T が lvalue の場合は lvalue の参照に、T が rvalue の場合は rvalue の参照として扱われます。

なお、Universal 参照はテンプレートのみではなく、C++11 で意味の変わった `auto` でも同様に使うことができます。

7. パーフェクトフォワーディング

> 次々　参照出る　まだまだ　授業続く
>
> 凍る　部屋の中
>
> 参照の左辺も右辺も　気にせず
>
> ゆっくりしていってね !!!

```
template<typename T>
void universal_pass(T&& a) {
  print_if_ref(a); // そのまま渡す
}

template<typename T>
void move_pass(T&& a) {
  print_if_ref(std::move(a)); // std::move で渡す
}

template<typename T>
void perfect_forwarding_pass(T&& a) {
  print_if_ref(std::forward<T>(a)); // std::forward で渡す
}
```

```
void perfect_forward() {
  int chirno = 9;
  // そのまま渡す。仮引数は lvalue として解釈される。
  universal_pass(chirno); // -> lvalue ref
  universal_pass(9); // -> lvalue ref

  // std::move で渡す。std::move では rvalue の参照にキャストするので、rvalue として渡される。
  move_pass(chirno); // -> rvalue ref
  move_pass(9); // -> rvalue ref

  // std::forward で渡す。
  // 受け取った変数が lvalue の参照か rvalue の参照かに応じて適切に関数に渡すことができる。
  perfect_forwarding_pass(chirno); // -> lvalue ref
  perfect_forwarding_pass(9); // -> rvalue ref
}
```

7.1. 解説

さて、ついにタイトルにもある perfect forwarding (完全転送) が出てきました。

Universal 参照で受け取った、lvalue か rvalue のどちらの参照かわからない変数を別の関数に渡す場合はどうしたら良いでしょうか。 そのまま渡せば良い様な気もしますが、 仮引数は lvalue として扱われてしまうため、rvalue の参照を受け取ったときも lvalue として渡してしまいますし、じゃあ `std::move` で渡せば良いかというと、今度は rvalue の参照にキャストしてしまうので lvalue の参照を受け取った場合でも rvalue の参照として渡してしまいます。

ここで出てくるのが `std::forward` です。`std::forward` は T の値を見て上手い具合にキャストをしてくれるので、受け取った変数が lvalue か rvalue のどちらの参照なのかに応じて、適切に渡すことができます。なお、`std::forward` の中身はテンプレートの特殊化で頑張って実装している様です。

8. まとめ

- もうバカでいいわよ！知らないっ！

不完全情報ゲームのナッシュ均衡を CFR (Counterfactual Regret Minimization) アルゴリズムで求めよう

ばいなり

1. はじめに

yabaitech.tokyo vol.7 を手に取ってくださりありがとうございます。前号の vol.6 では、世界的に見て最もポピュラーなポーカーの変種と言える**テキサスホールデム**の役判定を高速に行う話を書かせていただきましたが、今回も引き続いてポーカーを念頭に置いた記事を投稿したいと思います。

具体的には、 ポーカーを含む**不完全情報ゲーム**において**最適解**とも言える**ナッシュ均衡**を **CFR アルゴリズム**によって求めるプログラムの実装を目指します。もちろん、実際にナッシュ均衡が現実的な時間内に求まる（より正確には計算結果が十分収束する）かどうかはゲームの複雑さによるため、複雑性の非常に高いテキサスホールデムの完全解析を行おうと思うと本記事の内容ではまったく不可能ですが、それほど複雑でないゲームであれば最適解をプログラムによって求めることができるようになります。

ゲーム理論周りにあまり馴染みの無い方には、タイトルには仰々しい単語しか並んどらんしナッシュ均衡とか言われてもさっぱり分からんと思われてしまっているかもしれません。ただ、そういった方にもせめて雰囲気は伝わるような記事となるよう心掛けたので、ややタフな内容かもしれませんが各々の楽しみ方で読み進めていただけると幸いです[*2]。

2 と言うよりむしろ、筆者も特にゲーム理論の専門家でも何でもなくて、ゲーム理論を齧っただけの素人なりに初学者に興味を持ってもらうことも目的の一つにして記事を書きました。

2. 用語の説明

本章では、本記事のタイトルを構成する主なキーワードとなっている「不完全情報ゲーム」「ナッシュ均衡」「CFR アルゴリズム」の三点について、それぞれ説明をしていきます。

2.1. 不完全情報ゲーム

まず、タイトルの最初の単語である**不完全情報ゲーム**とは何ぞや？というところから見ていきましょう。簡単に言ってしまえば、ボードゲームなどのゲームにおいて、各々のプレイヤーがすべての情報にアクセスできるならばそのゲームは**完全情報ゲーム**、そうでないなら**不完全情報ゲーム**であると呼ばれます。

具体的な例を挙げると、**オセロ**や**将棋**といったゲームはお互いのプレイヤーが同じ盤面の情報（より正確には棋譜の情報）を共有していて、特定のプレイヤーからしかアクセスできないような情報は存在しないため、完全情報ゲームに分類されます。逆に、**ポーカー**や**麻雀**といったゲームは各々のプレイヤーが自分にしか分からない手札を持っていて、各プレイヤーがアクセスできる情報は同じではないため、不完全情報ゲームに分類されます。さらに、**じゃんけん**といった複数のプレイヤーが同時にアクションを起こす必要のあるゲームも不完全情報ゲームです。**バックギャモン**や**すごろく**といったゲームはどちらに分類すべきか微妙なところで、盤面がプレイヤーにすべて公開されているという観点では完全情報ゲームと言えますが、サイコロなどによるランダム性の解釈次第で不完全情報ゲームに分類されることもあります。これについては不確定完全情報ゲームという分類の仕方もあり、そちらの方がより正確とも言えますが、本記事ではこれらは不完全情報ゲームとして扱うことにしておきます。

完全情報ゲームでは、盤面が与えられると神の視点からはすでに勝敗（または引き分け）が定まっており、例えば 6×6 マスのオセロは初期盤面からお互いが最善を尽くすと後手が 4 石差で勝つことが知られています。この完全情報ゲームの攻略に関しては、完全読み切りが極めて困難であるような複雑なゲームに対しても、AlphaGo や AlphaZero といったプログラムがディープラーニングと強化学習を組み合わせた**深層強化学習**を用いて、盤面評価の精度の面でブレークスルーを最近引き起こしたことをご存じの方も多いでしょう。

不完全情報ゲームは、完全情報ゲームが持つこのような性質を持っていません。すべてのプレイヤーがそれぞれに与えられた情報をもとに「最善手」を取り続けても、さまざまな要因によって勝敗は変化しますし、そもそも「最善手」というのが特定の一手に定まらずに複数の候

補から確率的にアクションを選択するべき状況となることも少なくありません。また不完全情報ゲームにおいては、評価が可能なのは戦略であってアクションではないという点も非常に重要です。つまり、例えばじゃんけんにおいて「グーとチョキとパーを等確率で出す戦略」の良し悪しを評価することはできますが、「実際にグーを出した」というアクションを単体で評価することはできないということです。この性質がプログラムによる精度の高い盤面評価を難しいものにしています。

2.2. ナッシュ均衡

不完全情報ゲームについて説明したところで、次のキーワードである**ナッシュ均衡**についても説明していきましょう。ナッシュ均衡とは、ざっくり言ってしまえば「どのプレイヤーも利得（の期待値）をこれ以上増やすことができない状態」のことを指します。はじめにナッシュ均衡のことを「最適解」と呼びましたが、具体的にはこのような意味だったわけですね。

厳密な定義については記号などをちゃんと導入した後で再度確認しますが、もう少し正確に文章で記述すると「どのプレイヤーも、その他のプレイヤーがそのナッシュ均衡に従って戦略を選択している限りは、自分の戦略を変更することによって利得（の期待値）をこれ以上増やすことができないような戦略の組」のことをナッシュ均衡と言います。

男女の争い

簡単な例として**男女の争い**[*3]と呼ばれるゲームを見てみましょう。このゲームはある男女の組がデートに行こうとしており、一緒に行動したいという点では同意しているものの、お互いが行きたい場所が異なるという状況をモデル化したものです。ここでは男性はスポーツ観戦に、女性は映画鑑賞に行きたいということにしておきましょう。このとき、それぞれ男性と女性が得る利得は次のように定められるものとします：

男性 ＼ 女性	スポーツ観戦	映画鑑賞
スポーツ観戦	(2, 1)	(0, 0)
映画鑑賞	(0, 0)	(1, 2)

つまり、男性側は女性とともにスポーツ観戦に行った場合に最も利得を多く得ますが（２点）、

3 ポリティカルコレクトネス的に不適切な感のある名称・設定ですが、一般的に使用されている名称なのでその点についてはご容赦ください。ちなみに英語では “Battle of the sexes” となりますが、調べてみたところやはりポリティカルコレクトネスに配慮して “Bach or Stravinsky” ゲームと名称を変えている文献も少なくないようです。

次いで利得を多く得られるのは女性とともに映画鑑賞に行った場合で（１点)、一緒に行動できなかった場合は利得を得ることができません（０点)。

このような設定下におけるナッシュ均衡は、当たり前のようですが（男性 → スポーツ鑑賞 , 女性 → スポーツ鑑賞）と（男性 → 映画鑑賞 , 女性 → 映画鑑賞）の二組となります[*4]。男性側にとっては女性とともに映画鑑賞に行くというのは次善策なわけですが、女性側が映画鑑賞を選択している場合は、男性側の利得を最大化するためには男性側も映画鑑賞を選択する必要があるため、これが均衡の一つとなります。

この単純な例からナッシュ均衡の重要な性質を二点見出すことができます。一つは、一般にナッシュ均衡は必ずしも一組とは限らないということです。もう一つは、ナッシュ均衡はすべてのプレイヤーの戦略の組として与えられるため、ある一人のプレイヤーにとっての「相手の戦略によらない良い戦略」を与えてくれるようなものではないということです。

二人ゼロサムゲームとしてのじゃんけん

ところで、本記事では主にポーカーの戦略を考えたいわけですから、ゲームの条件として各プレイヤーの利得を足し合わせるとゼロになる、すなわちゲームが**ゼロサム**であることも追加しましょう。先ほどの男女の争いの例はゼロサムではありませんでしたので、ゼロサムであるゲームとして**じゃんけん**[*5]を考えてみることにしましょう。特に解説は要らないと思いますが、じゃんけんの利得表は次のようになります：

	グー	チョキ	パー
グー	(0, 0)	(1, -1)	(-1, 1)
チョキ	(-1, 1)	(0, 0)	(1, -1)
パー	(1, -1)	(-1, 1)	(0, 0)

このようなゲームにおいてもナッシュ均衡は存在するのでしょうか？ 例えばプレイヤー A が

4 後述する**混合戦略**を考慮すると本当はこの二組だけではないのですが、それに関してはまた後ほど。

5 知人が豪語していたじゃんけんの「必勝法」を紹介しましょう。曰く、「『最初はグー』の状態から遷移に最も時間が掛かるのはパーに切り替える場合だ。相手がパーに切り替えようとしているかどうかを見極めるには、相手の小指に注目すれば良い。よって、基本はこちらはグーのままだが、相手の小指が動いたらこちらはチョキに切り替えろ。グーからチョキに切り替えるのは、パーに切り替えるよりは素早く行えるから、優れた動体視力と反射神経があれば間に合わせることができる」。全員がこの「必勝法」を採用するとグーを出し続ける無限ループに陥ることになるので、そもそも「必勝法」の定義を満たしていないように筆者的には思うのですが、額面通りに事を運ばせられるのならば確かにまあ負けることは無いので、動体視力と反射神経に自信のある方は試してみると良いのではないでしょうか。

グーを出した場合を想定すると、プレイヤーBはパーを出すのが最善で、そうするとプレイヤーAはチョキを出すのが最善で …… と循環してしまい、お互いが妥協できる均衡状態は一見して存在せず、よってナッシュ均衡も存在しないように思えます。

ここで登場する概念が**混合戦略**というものです。混合戦略とは、例えばグーを50%、チョキを30%、パーを20%で出すといったように、確率的にアクションを決定する戦略のことです。なおこれに対して、例えば決まってグーを出すといったように確率によらない戦略は**純粋戦略**と呼ばれ、男女の争いの例では純粋戦略のみによるナッシュ均衡を紹介しました。今回のじゃんけんの例では、三種類の手をそれぞれ等確率に出す混合戦略を両者が採用する、というのがこのゲームにおける唯一のナッシュ均衡となります。

このナッシュ均衡の解釈は一筋縄ではいかないものです。三種類の手をそれぞれ等確率に出す混合戦略というのは、必ずグーを出すような非常に弱い相手に対しても利得の期待値がゼロにしかならない戦略です。ナッシュ均衡とはどのプレイヤーも利得をこれ以上増やすことができない状態のことで、言わば「最適解」であると説明されたはずなのに、そのような混合戦略を両者ともに取ることがそのナッシュ均衡なんです、と言われても納得できない方もいると思います。この事態を直感的に捉えられるようにするには、少なくとも筆者にとっては次のような言い換えを経る必要がありました：

（**二人ゼロサムゲーム**において一人がナッシュ均衡から外れたと仮定）
　ナッシュ均衡ではどのプレイヤーも利得をこれ以上増やすことができない
⇒ ナッシュ均衡から外れたプレイヤーは利得が増えることはない
⇒ **ゼロサム性**よりナッシュ均衡の戦略を取り続けたプレイヤーは利得が減ることはない
⇒ **ナッシュ均衡の戦略を取り続けることで利得の最低値が保証される**

つまり、**二人ゼロサムゲーム**においては、ナッシュ均衡の戦略を取ることは「相手の戦略によらず保証される利得を最大化できる」という意味で「最適解」なのであり、「相手の特定の戦略に対して利得を最大化できる」というようなものではないのです。なお後者のように特定の戦略を狙い撃ちするような戦略は**搾取戦略**と呼ばれ、例えば必ずグーを出す相手に対して必ずパーを出すといった戦略はこれに該当します。ただし、このように必ずパーを出すという戦略は、必ずチョキを出すという**対抗搾取戦略**に搾取されてしまいます。ナッシュ均衡とは、言わばすべてのプレイヤーがお互いを最大限搾取している理想的な状態で、達人どうしが戦ったらこうなりますという状態なわけですから、必ずパーを出すというような簡単に搾取されてしまう戦略

は、その理想からはかけ離れてしまっているわけですね。

このような考察を経ると、二人ゼロサムゲームにおいてはナッシュ均衡はまさに**「最善手」**の組であると言えることが分かります。ナッシュ均衡はあくまで戦略の組ですから、男女の争いの例を思い出していただくと分かるように、ナッシュ均衡に基づく戦略は一般のゲームにおいては「相手の戦略によらない良い戦略」であるとは限らなかったわけです。ところが、二人ゼロサムゲームにおいてはナッシュ均衡に基づく戦略を取ることで「相手の戦略によらず利得の最低値が保証される」わけですから、これはまさしく「最善手」であると言って良いでしょう。

なお、今回のじゃんけんの例で得られた「最善手」は三種類の手をそれぞれ等確率に出すというもので、これはどのような相手に対しても利得がゼロになってしまうというものでしたが、これはどちらかと言うと具体例が悪いという類のものです。よくあるポーカーの状況などでは、こちらがナッシュ均衡に基づく最善手を取っていて相手が最善手から離れているという場合、相手も最善手を取っている場合と比べて得られる利得は増加することがほとんどです。

補足

ところで、二人ゼロサムゲームにおけるナッシュ均衡の嬉しい性質については納得したが、三人以上が関与するゼロサムゲームではどうなのか、という点が気になっている方もいらっしゃるかもしれません。特に、テキサスホールデムなどは通常三人以上でテーブルを囲みますから、ポーカーを念頭に置いている本記事ではこの話題に触れておく必要があるでしょう。この点については、ふわっとした結論として「ナッシュ均衡に従うのは実用上はそれなりに強い場合が多いが、理論的な保証は無くなってしまう」と言うくらいが実は精々です。

ここで理論的な保証が無くなってしまうというのは、再び男女の争いの例と同じくナッシュ均衡に基づく戦略が「相手の戦略によらない良い戦略」とは限らないという意味です。これは三人ゼロサムゲームは二人非ゼロサムゲームの一般化である（ゼロサムとなるようにダミープレイヤーを一人追加すれば良い）ことを考えると分かるかもしれません。ただし、理論的な保証は失われると言っても、実用上はナッシュ均衡に従ってプレイするとそれなりに強いことが多く、三人以上が関与するゲームにおいてもナッシュ均衡を求めることが実際にはよく目標とされています。

また、これは理論寄りの話題となりますが、ほとんどすべての有限ゲームにはナッシュ均衡が奇数個存在することが知られています。じゃんけんについてはナッシュ均衡は一つだったので

良いとして、男女の争いについては中途半端に二つしか見つけられていませんが、どういうことなのでしょうか。実は、これは混合戦略を考慮していなかったことが原因で、混合戦略までちゃんと考えると（男性 → 2/3 の確率でスポーツ観戦 , 女性 → 2/3 の確率で映画鑑賞）という戦略の組もナッシュ均衡となります。

ここまで説明がやや長くなってしまいましたが、以上がナッシュ均衡についての説明でした。本記事ではナッシュ均衡を求めるプログラムをこれから語っていくことになるので、「ナッシュ均衡を求めることにどのような意味があるのか」についてあやふやなままだと記事としての説得力に欠けるかなあということで詳しめに説明してみましたが、いかがでしたでしょうか。

2.3. CFR (Counterfactual Regret Minimization) アルゴリズム

さて、ナッシュ均衡を乗り越えたと思ったらまたしても仰々しいキーワードが出てきてしまいましたね。CFR アルゴリズムとは、一言で言えば「展開型の二人ゼロサムゲームにおいてナッシュ均衡に収束することが保証されているアルゴリズム」なのですが、それがどのようなアルゴリズムなのか、まずは数式を使わずに文章で雰囲気を掴んでいただければと思います。

標準型ゲームと展開型ゲーム

CFR アルゴリズムの説明に入る前に、まずはゲームの分類についての話をしましょう。これまでに見てきた男女の争いやじゃんけんは、各プレイヤーが同時にアクションを起こすとそれぞれの利得が定まるという構造になっており、このようなゲームは**標準型ゲーム**（または**戦略型ゲーム**）と呼ばれます。これに対して、テキサスホールデムのように**手番**が存在するゲームは**展開型ゲーム**と呼ばれます。展開型ゲームは、ゲームの状態を表現する頂点およびアクションによって遷移できることを示す有向辺からなる**ゲーム木**を用いて、グラフ形式で数学的に表現されます。また不完全情報ゲームにおいては、実際には異なる状態だがあるプレイヤーからはその区別がつかないという状況がよく発生するでしょう。このような状況に対応するために、展開型ゲームでは各プレイヤーが区別できない状態について定義する**情報分割**も与えられます。

なお、すべての標準型ゲームは展開型ゲームとして表現することもできます。具体的には、便宜上の手番を順番に割り振って、手番が後のプレイヤーはそれまでになされたアクションの区別がつかない、という設定にすれば良いだけです。例えばじゃんけんならば、あるプレイヤーがグーを出すことを予め決定しているものの、他のプレイヤーからはそれが観測できないというような状況に対応しています。

リグレットと regret-matching アルゴリズム

それでは、続いてキーワード中の**リグレット (regret)** の部分の説明をしていきましょう。リグレットとは直訳すると**後悔**のことで、日本語訳する際にリグレットと呼ばずに後悔という語がそのまま使われることもありますが、これは過去の自分のアクションに対して「あのときああすれば良かったのに」という後悔の度合いを定量化したものです。

標準型ゲームであるじゃんけんを再び例にとって考えます。自分はグーを出し、相手にパーを出されて負けた（１点を失った）としましょう。このとき、もし自分もパーを出していれば引き分けとなって１点を失わずに済んだはずだったので、パーに対するリグレットは１点となります。もっと言えば、自分がチョキを出せていれば勝ちとなり、むしろ１点を得ることが出来ていたはずだったので、チョキに対するリグレットは差し引き２点となります。なお、もし逆に自分がパーを出し、相手はグーを出して勝負に勝った（１点を得た）場合、グーとチョキに対するリグレットはそれぞれ－１点と－２点になり、リグレットは負の値を取ることになります。

さて、じゃんけんも何戦かこなすと、これまでのリグレットの総和を計算することで「過去にどのようなアクションを取るべきだったのか」がだんだんと見えてくることになるでしょう。このリグレットの総和（ただし総和が負の値になった場合はゼロとして扱う）に比例するように混合戦略を立てて将来的なリグレットを最小化しよう、という単純かつ強力な手法が **regret-matching アルゴリズム**と呼ばれるものです。具体的には、例えばグー、チョキ、パーそれぞれに対する現在のリグレットの総和が２点、－１点、８点であるとしましょう。このとき、負の値となっているチョキに対するリグレットの総和は０点として扱うこととし、２点、０点、８点に比例するように、グー、チョキ、パーをそれぞれ 20%、0%、80% の確率で出す混合戦略を立てるということになります。

この regret-matching アルゴリズムを用いて**自己対戦**を繰り返し行うと、各時刻の混合戦略の平均[*6]が**相関均衡**に収束することが知られています[*7]。なお、ここで相関均衡とかいう用語が突然出てきてしまいましたが、これはナッシュ均衡を一般化した概念で、今回の二人じゃんけんの例ではナッシュ均衡と一致するので特に気にしないことにしましょう。この枠組みを展開型ゲームに拡張したのが続いて見ていく CFR アルゴリズムになります。

---

6 時刻が無限大のときの混合戦略そのものではない点に注意。

7 S. Hart and A. Mas-Colell. **A simple adaptive procedure leading to correlated equilibrium.** Econometrica, 68(5):1127–1150. 2000.

Counterfactual regret

さて、ここまで説明してようやく本題の **CFR (Counterfactual Regret Minimization) アルゴリズム**の説明に移ることができます。と言っても実はすでに説明の大半は終わっていて、先ほどの regret-matching アルゴリズムにおいて **counterfactual regret** と呼ばれるリグレットを用いるというだけなのですが、この counterfactual regret が曲者なので、その定義に必要な counterfactual 到達確率および counterfactual value と併せて軽く説明します。

ここでの目標は、「事実に反する」「反事実的」という意味の “counterfactual” という語がなぜ用いられているのかを説明することに留めます。というのも、counterfactual regret をちゃんと理解しようと思うと数式を避けて通れないためで、正確な定義は次章を参照してください。

まず、各プレイヤーの戦略が固定されているものとします。このとき、アクションの**履歴** h に対するプレイヤー i の **counterfactual 到達確率**を「プレイヤー i の戦略の寄与を無視して計算されたアクションの履歴 h の実現確率」として定義します。つまり、プレイヤー i が行ってきた確率的選択は counterfactual 到達確率には一切影響せず、プレイヤー i 以外の寄与のみを取り出して計算するということです。このように事実に反した計算を行うので、“counterfactual” であると呼ばれるわけですね。

いま、利得の値は本来はゲームの終了状態においてのみ定まる値ですが、各プレイヤーの戦略が固定されているので、現在の履歴 h におけるプレイヤー i の利得の期待値をボトムアップに計算することができます。この利得の期待値と counterfactual 到達確率の積を **counterfactual value** と呼びます。最後に、現在の履歴 h における **counterfactual regret** とは、利得の値を直接使う代わりにこの counterfactual value を用いて計算されたリグレットのことです。

この counterfactual regret による regret-matching アルゴリズムを用いて自己対戦を繰り返し行うと、展開型の二人ゼロサムゲームにおいては各時刻の戦略の重み付き平均がナッシュ均衡に収束します[*8][*9]。今回も説明が長くなってしまいましたが、これが CFR アルゴリズムと呼ばれるものの正体で、ポーカーを含む不完全情報ゲームの近年の解析における基礎的なアルゴリズムの一つとなっています。

---

8 M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione. Regret minimization in games with incomplete information. In NIPS, pages 1729–1736. 2007.

9 ナッシュ均衡の説明に引き続いてまたしても「二人ゼロサム」の条件が加えられてしまいましたが、今回も三人以上の設定でも多くの場合は「実用上それなりに強い」解を求められることが知られています。

補足：ナッシュ均衡を厳密に求めるアルゴリズム

さて、ここまでの CFR アルゴリズムの説明をよく読むと、ナッシュ均衡に「収束」するという表現が用いられていて、ナッシュ均衡解が「求まる」とは表現されていませんでした。これは、CFR アルゴリズムは**反復解法**であるため、数学的な意味でナッシュ均衡を厳密に満たす戦略の組を有限の時間で求められるようなアルゴリズムではないためです。なお、反復解法とは定められた処理を繰り返すことで求めたい解の近似を徐々に改善していく手法のことです。

ナッシュ均衡を**解析的**に求めたい、すなわち数学的にナッシュ均衡を厳密に満たす戦略の組を求めたいという場合は、**直接解法**を用いる必要があります。実際、二人ゼロサムゲームならば**線形計画法**によって多項式時間でナッシュ均衡解を厳密に求めることができます[*10]。二人非ゼロサムゲームおよびその一般化の三人ゼロサムゲームにおいては、多項式時間のアルゴリズムが存在するかどうかは未解決問題で、もしそのようなアルゴリズムが存在した場合は計算量理論において大きな意味を持つようです[*11]（ので存在しなそうということになります）。

それならば、二人ゼロサムゲームに対しては CFR アルゴリズムを使わずに線形計画法で解いてしまえば良いのではと思われるかもしれませんが、実際には線形計画法は多項式時間で動作すると言っても解ける問題の大きさは限られています。CFR アルゴリズムを本記事で紹介しているのはそのためで、実用上はこちらの方がより大きい問題に対応することができます。

3. 数学的な定義

本記事のタイトルを構成するキーワードについての大雑把な説明が済んだところで、数学的な議論を行えるようにいよいよ記号を導入していきましょう。定義が突然大量に出てきて混乱してしまうかもしれませんが、どれも今後の議論で必要となる定義ですので、必要に応じてここに立ち返ってきてください。

- $N \coloneqq \{0, 1, \cdots, n-1\}$: n 人のプレイヤー全体の集合。
- c: **偶然手番**。アクションを予め定められた確率に基づいて実行する仮想的なプレイヤー。サイコロを振る、カードをシャッフルする、といった行為はこの偶然手番が行っているものと見なします。

10 ただし、展開型ゲームにおいては**完全記憶 (perfect recall)** という性質がゲームに備わっている必要があります。完全記憶でない展開型ゲームのナッシュ均衡を求めるのは NP 困難です。

11 PPAD という計算量クラスがあるようで、これらの問題は PPAD 完全であるとのことです。

- $h \in H$: **履歴** h および履歴全体の集合 H. 履歴とはこれまでになされたすべてのアクションの列のことで、ゲーム木における頂点に対応します。盤面などが同じであっても履歴が異なる場合はそれらを区別するということを明示するためにも、ゲームの「状態」などとは呼ばずに「履歴」という語を用います。
- $Z \subseteq H$: **終端履歴**の集合。終端履歴 $z \in Z$ まで到達するとゲームは終了し、各プレイヤーに対する利得が定まります。
- $P : H \setminus Z \to N \cup \{c\}$: **手番関数**。$P(h)$ は非終端履歴 $h \in H \setminus Z$ において次にアクションを行うプレイヤーを表します。
- $A(h) \coloneqq \{a \mid (h, a) \in H\}$: **アクション集合**。$A(h)$ は非終端履歴 $h \in H \setminus Z$ においてプレイヤー $P(h)$ が行うことが可能なアクションの集合を表します。ここで、(h, a) は履歴 h の後にアクション a を行ったときの履歴を表します。
- $u_i : Z \to \mathbb{R}$: プレイヤー i の**利得関数**。$u_i(z)$ は、終端履歴 $z \in Z$ における、プレイヤー i にとっての利得の実数値を表します。二人ゼロサムゲームの場合 $u_0(z) = -u_1(z)$ が成り立ちます。
- $I_i \in \mathcal{I}_i$: プレイヤー i に関する**情報集合** I_i および**情報分割** $\mathcal{I}_i$. なおプレイヤーに関して興味が無い場合は、添字の i を省略することがあります。プレイヤー i に関する情報集合 I_i は、手番がプレイヤー i であるような履歴の集合のことで、$h, h' \in I_i$ のとき、つまり履歴 h と h' がともに同じ情報集合 I_i に属するとき、プレイヤー i からはこれらの履歴を区別することができないことを意味します。また、すべての履歴は必ずただ一つの情報集合に属します。さらに、手番関数 P やアクション集合 A について、$h, h' \in I$ ならば $P(h) = P(h')$ および $A(h) = A(h')$ が成り立つので、これらを $P(I)$, $A(I)$ と型を混同して記述することにします。
- $\sigma_i \in \Sigma_i$: プレイヤー i の**戦略** σ_i および戦略全体の集合 Σ_i. 戦略 σ_i は、情報集合 I_i を受け取って次に行うアクション候補の集合 $A(I_i)$ 上の確率分布を返す関数です。また、戦略 σ_i のもとで情報集合 I_i においてアクション a を行う確率を $\sigma_i(I_i, a)$ と表記します。なお、$\sigma_i(I_i)$ は確率分布なので $\sum_{a \in A(I_i)} \sigma_i(I_i, a) = 1$ が成り立ちます。さらに、各プレイヤーの戦略の組 $\{\sigma_0, \sigma_1, \cdots, \sigma_{n-1}\}$ を σ と表記し、σ から σ_i を除いたものを σ_{-i} と表記します。
- $\pi^\sigma : H \to [0, 1]$: $\pi^\sigma(h)$ は、各プレイヤーが戦略の組 σ に従ったときの履歴 h への**到達確率**を表します。また、$\pi_i^\sigma(h)$ を $\pi^\sigma(h)$ におけるプレイヤー i の寄与として定めます。すなわち、$\pi^\sigma(h) = \prod_{i \in N \cup \{c\}} \pi_i^\sigma(h)$. さらに、$\pi_{-i}^\sigma(h)$ を $\pi^\sigma(h)$ からプレイヤー i の寄与 $\pi_i^\sigma(h)$ を除外した到達確率として定めます。すなわち、$\pi_{-i}^\sigma(h) \coloneqq \pi^\sigma(h) / \pi_i^\sigma(h)$.

- $\pi^\sigma : \mathcal{I} \to [0, 1]$: $\pi^\sigma(I)$ は、各プレイヤーが戦略の組 σ に従ったときの、情報集合 I に含まれるいずれかの履歴への到達確率を表します。すなわち、$\pi^\sigma(I) \coloneqq \sum_{h \in I} \pi^\sigma(h)$. また、$\pi_i^\sigma(I)$ および $\pi_{-i}^\sigma(I)$ もこれまでと同様に定義します。
- $u_i^\sigma(h) \coloneqq \sum_{z \in Z} \pi^\sigma(h \to z) u_i(z)$: 各プレイヤーが戦略の組 σ に従ったときの、履歴 h におけるプレイヤー i の**利得の期待値**。ここで、$\pi^\sigma(h \to z)$ は履歴 h に到達した状態から、各プレイヤーが戦略の組 σ に従って履歴 z に到達する確率を表します。また、$u_i(\sigma)$ をゲーム開始時におけるプレイヤー i の利得の期待値として定義します。すなわち、$\varnothing$ をゲーム開始時の履歴としたとき、$u_i(\sigma) \coloneqq u_i^\sigma(\varnothing)$.

3.1. ナッシュ均衡の定義

記号をちゃんと定義したので、ナッシュ均衡についてまずは定義を確認してみましょう。前章では、ナッシュ均衡を「どのプレイヤーも、その他のプレイヤーがそのナッシュ均衡に従って戦略を選択している限りは、自分の戦略を変更することによって利得の期待値をこれ以上増やすことができないような戦略の組」であると説明しました。これを数式を用いて記述すると、戦略の組 σ^* が**ナッシュ均衡**であるとは、すべてのプレイヤー i に対して

$$u_i(\sigma^*) = \max_{\sigma_i \in \Sigma_i} u_i\left(\sigma_i, \sigma^*_{-i}\right)$$

が成り立つことであると表現できます。ここで、$u_i\left(\sigma_i, \sigma^*_{-i}\right)$ は $u_i\left(\left\{\sigma_i\right\} \cup \sigma^*_{-i}\right)$ の略記とします。

また、ナッシュ均衡を緩和した **ε- ナッシュ均衡**を導入します。前章で説明したように CFR アルゴリズムは反復解法であり、ナッシュ均衡を厳密に満たす戦略の組を有限の時間で求められるものではないため、ナッシュ均衡の近似という概念を正確に定義したいためです。ε- ナッシュ均衡を以下のように定義することで、これまで「ナッシュ均衡に収束する」とぼんやり表現していたものが、「ε が 0 に収束する」という意味に正確に定まります。

それでは ε- ナッシュ均衡の定義ですが、$\varepsilon \geq 0$ に対して、戦略の組 σ^* が **ε- ナッシュ均衡**であるとは、すべてのプレイヤー i に対して

$$u_i(\sigma^*) + \varepsilon \geq \max_{\sigma_i \in \Sigma_i} u_i\left(\sigma_i, \sigma^*_{-i}\right)$$

が成り立つことを言います。つまり、こちらは自分の戦略を変更することでちょっと $(\leq \varepsilon)$ だけ利得の期待値を増やせる余地が残っているような戦略の組を許容しています。ε が小さいほどより良いナッシュ均衡の近似であると言え、特に $\varepsilon = 0$ のときはナッシュ均衡と一致します。

可搾取量

さて、それでは実際に CFR アルゴリズムを実行して得られた戦略の組に対応する ε の値を見積もるにはどうしたら良いでしょうか。その上界を与えてくれるのが**可搾取量 (exploitability)** と呼ばれる値です。

可搾取量を定義する前に、まず**最適反応戦略**を定義しましょう。プレイヤー i 以外の戦略の組 σ_{-i} が与えられたときのプレイヤー i の最適反応戦略 $b_i(\sigma_{-i})$ は次のように定められます：

$$b_i(\sigma_{-i}) \coloneqq \arg\max_{\sigma_i \in \Sigma_i} u_i(\sigma_i, \sigma_{-i}).$$

つまり、最適反応戦略とはその名の通り、σ_{-i} が定まっているもとでプレイヤー i の利得を最大化するような戦略のことです。最適反応戦略は複数存在する場合もありますが、そのような場合はどの戦略を取ってきても良いものとします。このように最適反応戦略を定義すると、戦略の組 σ^* がナッシュ均衡であるとき、すべてのプレイヤー i に対して

$$\sigma_i^* = b_i(\sigma_{-i}^*)$$

が成り立つように $b_i(\sigma_{-i}^*)$ を取ることができます。

ここで、ゲームを二人ゲームに再び限定することにします。このとき、プレイヤー i の戦略 σ_i の**可搾取量** $\varepsilon_i(\sigma_i)$ を次のように定義します：

$$\varepsilon_i(\sigma_i) \coloneqq u_i(\sigma^*) - u_i\big(\sigma_i, b_{-i}(\sigma_i)\big).$$

なお、ここで σ^* はナッシュ均衡を満たす戦略の組とします。つまり戦略 σ_i の可搾取量とは、こちらの戦略が相手に筒抜けで常に最適反応戦略を取られてしまうような状況において、こちらがナッシュ均衡に基づく最適戦略を取っていた場合と比べて、現在の戦略 σ_i が相手にどれだけ搾取されてしまうかを表す値のことです。

最後に、戦略の組 σ に対して定まる**可搾取量** $\varepsilon(\sigma)$ を次のように定義します：

$$\varepsilon(\sigma) \coloneqq \varepsilon_0(\sigma_0) + \varepsilon_1(\sigma_1) = u_0\big(b_0(\sigma_1), \sigma_1\big) + u_1\big(\sigma_0, b_1(\sigma_0)\big).$$

つまり、戦略の組 σ の可搾取量は二人ゲームにおいては両プレイヤーの戦略の可搾取量の和として定義されます。なお、右側の等号ではゼロサム性より $u_0(\sigma) = -u_1(\sigma)$ が成り立つことを利用しており、特に $u_i(\sigma^*)$ の値は一般には未知ですがキャンセルされます。特に証明は行いませんが、このように可搾取量を定義すると $\varepsilon(\sigma) \leq \varepsilon$ のとき、戦略の組 σ は少なくとも ε- ナッシュ均衡であると言えます。特に、戦略の組 σ^* がナッシュ均衡のときは $\varepsilon(\sigma^*) = 0$ が成り立ちます。

3.2. CFR アルゴリズムの定義

文章だけの説明ではいまいち良く分からなかった CFR アルゴリズム周りの定義についても引き続き確認していきましょう。前章の説明において **counterfactual 到達確率**と呼んでいたものは、まさに先ほど定義した $\pi^{\sigma}_{-i}(h)$ のことです。続いて、**counterfactual value** はこの counterfactual 到達確率と利得の期待値の積のことでしたから、次のように定義されます：

$$v_i(\sigma, h) \coloneqq \pi^{\sigma}_{-i}(h) \cdot u^{\sigma}_i(h).$$

履歴 h におけるアクション a に対する **counterfactual regret** は、次のように定義されます：

$$r_i(h, a) \coloneqq v_i\big(\sigma_{I \to a}, h\big) - v_i(\sigma, h).$$

ここで $\sigma_{I \to a}$ は、履歴 h を含むような情報集合 I を受け取った場合のみ、必ずアクション a を取るように戦略 σ を変更したものを表すこととします。さらに、情報集合 I におけるアクション a に対する **counterfactual regret** を次のように定義します：

$$r_i(I, a) \coloneqq \sum_{h \in I} r_i(h, a).$$

CFR アルゴリズムは自己対戦を繰り返し行うというものでしたから、時刻 t における戦略 σ^t のもとでの counterfactual regret を $r^t_i(I, a)$ で表すことにしましょう。このとき、counterfactual regret の累積値 $R^T_i(I, a)$ は単に次のように与えられます：

$$R^T_i(I, a) \coloneqq \sum_{t=1}^{T} r^t_i(I, a).$$

各時刻における戦略 σ^t は、regret-matching アルゴリズムによってこれまでのリグレットの総和に比例するように与えられることを思い出すと、次のように表すことができます：

$$\sigma^{T+1}_i(I, a) \coloneqq \begin{cases} \dfrac{\left[R^T_i(I, a)\right]^+}{\displaystyle\sum_{a' \in A(I)} \left[R^T_i(I, a')\right]^+} & \text{if } \displaystyle\sum_{a' \in A(I)} \left[R^T_i(I, a')\right]^+ > 0 \\ 1 \,/\, \big|A(I)\big| & \text{otherwise.} \end{cases}$$

なお、ここで $[x]^+ = \max(x, 0)$ です。最後に、二人ゼロサムゲームにおいてナッシュ均衡に収束するのは各時刻の戦略を $\pi^{\sigma^t}_i(I)$ で重み付けした場合の平均戦略です：

$$\bar{\sigma}^T_i(I) \coloneqq \frac{\displaystyle\sum_{t=1}^{T} \Big(\pi^{\sigma^t}_i(I) \cdot \sigma^t_i(I)\Big)}{\displaystyle\sum_{t=1}^{T} \pi^{\sigma^t}_i(I)}.$$

CFR アルゴリズムの改良

以上が「バニラ」な CFR アルゴリズムの定義ですが、収束を実用上早めるためにさまざまな変種が提案されてもいます。ここでは、実装が比較的簡単でかつ効果の高い **Discounted CFR (DCFR) アルゴリズム**[*12]を紹介することにしましょう。

DCFR アルゴリズムは 3 つの実数パラメータ α, β, γ を取り、先ほどの $R_i^T(I, a)$ と $\bar{\sigma}_i^T(I)$ を次のように再定義します：

$$R_i^{T+1}(I, a) := \begin{cases} \dfrac{T^\alpha}{T^\alpha + 1} R_i^T(I, a) + r_i^{T+1}(I, a) & \text{if } R_i^T(I, a) \geq 0 \\ \dfrac{T^\beta}{T^\beta + 1} R_i^T(I, a) + r_i^{T+1}(I, a) & \text{otherwise,} \end{cases}$$

$$\bar{\sigma}_i^T(I) := \frac{\sum_{t=1}^{T} \left(t^\gamma \cdot \pi_i^{\sigma^t}(I) \cdot \sigma_i^t(I) \right)}{\sum_{t=1}^{T} \left(t^\gamma \cdot \pi_i^{\sigma^t}(I) \right)}.$$

$(\alpha, \beta, \gamma) = (\infty, \infty, 0)$ のとき、バニラな CFR アルゴリズムと一致します（$T + 1 = 2$ のときに変なことが起こりますがそれは無視することとして）。提案論文では、さまざまなパラメータを実験的に試した結果 $(\alpha, \beta, \gamma) = (1.5, 0, 2)$ が一貫して良いパフォーマンスを示すとされており、今後の実装においてもこのパラメータを採用することとします（特に、$(\alpha, \beta, \gamma) = (\infty, -\infty, 1)$ の場合に相当する **CFR+ アルゴリズム**[*13]がそれまでの state-of-the-art とされていたのですが、この CFR+ に対して一貫してより優れた結果をもたらしています）。

非常にアドホックな感がある改変ではありますが、なぜこれで上手くいくようになるのかをかなりざっくり説明すると、リグレットの累積値 $R_i^T(I, a)$ および平均戦略 $\bar{\sigma}_i^T(I)$ の計算において、各時刻における値を均一に重み付けするのではなく、後の時刻ほど重みが大きくなるように改変をしている点が本質です。つまり、最初の方の時刻の寄与が “discount” されているわけですね。時刻が後になるほど徐々に賢い戦略を獲得していくアルゴリズムですから、後の時刻ほど重みを大きく設定することで収束を早くできる、というのは直感的にも正しいように思いますが、いかがでしょうか。

---

12 N. Brown and T. Sandholm. Solving imperfect-information games via discounted regret minimization. In AAAI, pages 1829–1836. 2019.

13 O. Tammelin, N. Burch, M. Johanson, and M. Bowling. Solving heads-up limit texas hold'em. In IJCAI, pages 645–652. 2015.

4. CFR アルゴリズムの実装

さて、 ようやく本章からは皆さんお待ちかねの実装パートです。 プログラムの記述言語には実行速度と実装のエレガントさを重視して Rust を採用することとします。yabaitech を手に取っていただけるような方々には Rust もお茶の子さいさいでしょう、…… というのは半分冗談にしても、いくつかの言語に触れたことがある方ならばプログラムを読むのにそれほど困りはしないと思います。

これから紹介するプログラムは、`https://github.com/b-inary/yabai-vol7-src` にアクセスすることで完全な形を確認することもできます。

4.1. ゲームのインターフェース定義

まずは CFR アルゴリズム本体の実装に移る前に、ゲームを表現するインターフェースを先に定義することにしましょう。CFR アルゴリズムをこのインターフェースのもとで実装することで、具体的なゲームを考える際にはインスタンス化を行うだけで CFR アルゴリズムを適用できるようになります。

本記事では主にポーカーを念頭に置いているので、ゲームの構造を制限してポーカーに特化した最適化を行えるようにします。具体的には、手札を配るといったプライベートな情報を与えるイベントはゲームの最初にしか発生しないという条件を加えます。例えばテキサスホールデムを考えると、最初に全員に二枚の手札が配られた後はコミュニティカードの公開も含めて全員が正しくすべてのアクションを把握できるという構造になっていることが分かります。本記事ではこのように全員が把握できるアクションは**パブリック**であると呼ぶことにしましょう。また、実装を簡単にするためにゲームは二人ゲームに限定することとし、各プレイヤーが持つ手札の構造は対称であるものとします。

ゲームにこのような条件を加えるとどのような最適化が行えるかというと、本来ならばゲームの最初に偶然手番がアクションを行うため、その分だけゲーム木も枝分かれするはずなのですが、これを飛ばしてパブリックなアクションの履歴のみを追跡し、終端履歴に到達したときに初めて最初に遡ってすべての手札の可能性をまとめて計算する、ということが行えるようになります。ただし、このような最適化を施すことで実装の読みやすさは犠牲になる部分があるので、読みづらいと思われる点はその都度解説していこうと思います。

それでは具体的な実装に移っていきますが、まずはゲームのインスタンスが満たすべき振る舞いを定めるトレイト（インターフェース）を次のように定義します：

```
/// ゲームの定義を表すインターフェース
pub trait Game {
    /// ゲーム木のノードを表す型
    type Node: GameNode;

    /// ゲーム木の根、すなわちゲームの初期履歴を返す
    fn root() -> Self::Node;

    /// プライベートな手札の組み合わせの個数を返す
    fn num_private_hands() -> usize;

    /// 終端履歴 `node` において、最初の偶然手番の寄与を含まない counterfactual 到達確率が
    /// `pmi` のときの `player` の counterfactual value を計算する
    fn evaluate(&self, node: &Self::Node, player: usize, pmi: &Vec<f64>) -> Vec<f64>;
}
```

この Game トレイトは、 そのインスタンスに対して Node, root(), num_private_hands(), evaluate() の四点の定義を要求していることが読み取れます。このうち、Node は GameNode トレイトを満たす型で、その GameNode トレイトの詳細については後述します。また、root() および num_private_hands() は、それぞれゲームの定義に関わる関数です。

最後の evaluate() については、いきなり結構な難所なので詳しめに説明します。コードとコメントを読むと、evaluate() は &self の他に三個の引数 node, player, pmi を受け取って「終端履歴 node において、最初の偶然手番の寄与を含まない counterfactual 到達確率が pmi のときの player の counterfactual value を計算する」というメソッドであることが分かります。ここで、counterfactual value は counterfactual 到達確率と利得の期待値の積

$$v_i(\sigma, h) \coloneqq \pi^{\sigma}_{-i}(h) \cdot u^{\sigma}_i(h)$$

で与えられるのでした。また終端履歴においては、利得の期待値は単に利得の値そのものと一致します。よって、このメソッドは node における player の利得を計算し、さらに引数として受け取った pmi を掛け合わせた結果を返すことを期待されていることが分かります。

evaluate() が counterfactual value を返すのは良いとしても、関数の返り値と引数 pmi の

型が実数の配列となっているのが、 このメソッドの定義を理解する上でもう一つ厄介な点となっています。 この点についても説明していくことにすると、 まずこれらの配列の要素数は `num_private_hands()` の値と一致することが期待されています。 また、 返り値の配列の各要素は `player` が特定の手札を持っているときの counterfactual value を表します。 引数の counterfactual 到達確率 `pmi` については、 相手（正確には自分以外）のプレイヤーが持っている手札に応じて値が定まるものなので、`pmi` の各要素には相手のプレイヤーが特定の手札を持っているときの counterfactual 到達確率が入っています。

引き続いて、各アクションを $0, 1, \cdots, \left|A(h)\right| - 1$ の整数に対応させて表現することとし、パブリックな履歴をアクションの配列で表現することにしましょう。

```
/// アクションを表す型
pub type Action = usize;

/// パブリックな履歴を表す型
pub type PublicHistory = Vec<Action>;
```

さらに、先ほど出現した、ゲーム木のノードを表現するインターフェースである `GameNode` トレイトを次のように定義します：

```
/// ゲーム木のノードを表すインターフェース
pub trait GameNode {
    /// 現在のパブリックな履歴を返す
    fn public_history(&self) -> &PublicHistory;

    /// 現在のノードが終端履歴かどうかを返す
    fn is_terminal(&self) -> bool;

    /// 現在の手番のプレイヤーを返す
    fn current_player(&self) -> usize;

    /// 着手可能なアクションの個数を返す
    fn num_actions(&self) -> Action;
```

```
    /// 着手可能なアクションの一覧を返す
    fn actions(&self) -> std::ops::Range<Action> {
        0..self.num_actions()
    }

    /// `action` を行った後のノードを返す
    fn play(&self, action: Action) -> Self;
}
```

この GameNode トレイトのメソッドシグネチャは素直なものばかりで、特に理解が難しい点は無いと思います。

4.2. CFR アルゴリズムの実装

それではゲームを表現するインターフェースが定まったところで、続いて CFR アルゴリズム本体の実装にいよいよ取り掛かっていきましょう。トップダウンに大枠から細部の順に実装を紹介していくことにします。

CFR アルゴリズムを管理する構造体の定義

まずは、CFR アルゴリズムを管理する構造体の定義から見ていきましょう。

```
use std::collections::HashMap;

pub struct CFRMinimizer<'a, T: Game> {
    /// ゲーム定義のインスタンス
    game: &'a T,

    /// リグレットの累積値
    cum_regret: HashMap<PublicHistory, Vec<Vec<f64>>>,

    /// 各時刻の戦略の和
    cum_strategy: HashMap<PublicHistory, Vec<Vec<f64>>>,

    /// Discounted CFR のパラメータ
```

```
    alpha_t: f64,

    /// Discounted CFR のパラメータ
    beta_t: f64,

    /// Discounted CFR のパラメータ
    gamma_t: f64,
}
```

Rust では構造体の定義にはフィールドのみが含まれ、メソッドは `impl` ブロックに記述することになるので、とりあえずはこのフィールドたちを眺めることにしましょう。Rust の大きな特徴の一つでもある参照のライフタイムの管理に必要な注釈 `'a` がちょっとうるさいですが、まず `game` は先ほどの `Game` トレイトを満たすインスタンスへの参照を保持するフィールドで、この `game` に対して CFR アルゴリズムが実行されます。後ろのフィールド `alpha_t`, `beta_t`, `gamma_t` については非本質的なので省略します。残った `cum_regret` と `cum_strategy` についてはもう少し詳しい説明をしましょう。

これら二個のフィールドの型は `HashMap<PublicHistory, Vec<Vec<f64>>>` となっています。それぞれリグレットの累積値と各時刻の戦略の和を管理している変数ですが、パブリックな履歴がハッシュマップのキーとなっているのは良いものの、それで返ってくる値が二次元配列となっているのはどういうことなのでしょうか。まず外側の配列は分かりやすくて、リグレットや戦略はアクションに対して定まるものですから、単に各アクションを受け取ります。それで残った方の配列ですが、これは先ほどと同じで手番となっているプレイヤーが特定の手札を持っているときの値が要素となっています。つまり、`cum_regret[public_history][action][private_hand]` のようにアクセスすると値を引くことができます。

さて、ここからは `impl` ブロック内の記述に移っていくことになります。これも非本質的ですが、定数 `ALPHA`, `BETA`, `GAMMA` とコンストラクタを今のうちに定めてしまうことにします：

```
impl<'a, T: 'a + Game> CFRMinimizer<'a, T> {
    const ALPHA: f64 = 1.5;
    const BETA: f64 = 0.0;
    const GAMMA: f64 = 2.0;
```

```
    /// コンストラクタ
    pub fn new(game: &'a T) -> Self {
        Self {
            game,
            cum_regret: HashMap::new(),
            cum_strategy: HashMap::new(),
            alpha_t: 1.0,
            beta_t: 1.0,
            gamma_t: 1.0,
        }
    }

    ...
```

CFR アルゴリズムの枠組みの実装

続いて、CFR アルゴリズムを実行する本体となるメソッドを書いていきます。アルゴリズムの大枠は自己対戦を繰り返すというものですが、各時刻における複雑な処理の部分はとりあえず `cfr_recursive()` に任せてしまうことにすると、実装はおおよそ次のようになるでしょう：

```
/// CFR アルゴリズムによる学習を行い、平均戦略を返す
pub fn compute(
    &mut self,
    num_iterations: i32,
) -> HashMap<PublicHistory, Vec<Vec<f64>>> {
    // ゲームの初期履歴を取得
    let root = T::root();

    // ゲーム木を構築して累積値を 0 で初期化
    Self::build_tree(&root, &mut self.cum_regret);
    Self::build_tree(&root, &mut self.cum_strategy);

    // 到達確率を 1 で初期化
    let ones = vec![1.0; T::num_private_hands()];
```

```
    // 自己対戦を繰り返す
    for t in 0..num_iterations {
        let t_f64 = t as f64;
        self.alpha_t = t_f64.powf(Self::ALPHA) / (t_f64.powf(Self::ALPHA) + 1.0);
        self.beta_t = t_f64.powf(Self::BETA) / (t_f64.powf(Self::BETA) + 1.0);
        self.gamma_t = (t_f64 + 1.0).powf(Self::GAMMA);

        // プレイヤー毎に処理を行う
        for player in 0..2 {
            self.cfr_recursive(&root, player, &ones, &ones);
        }
    }

    self.compute_average_strategy()
}
```

まず、build_tree() はゲーム木を構築して cum_regret および cum_strategy を初期化する関数で、次のような特に工夫の要らない再帰関数として定義されます：

```
/// ゲーム木を構築する
fn build_tree(node: &T::Node, tree: &mut HashMap<PublicHistory, Vec<Vec<f64>>>) {
    if node.is_terminal() {
        return;
    }
    tree.insert(
        node.public_history().clone(),
        vec![vec![0.0; T::num_private_hands()]; node.num_actions()],
    );
    for action in node.actions() {
        Self::build_tree(&node.play(action), tree);
    }
}
```

さて、cum_regret および cum_strategy を初期化したらいよいよ自己対戦を繰り返すことになります。ここで、何となくさらっと流されやすい点ですが、各時刻においてプレイヤー毎に

関数 cfr_recursive() による処理を行っている点は注意が必要です。

関数 cfr_recursive() の仕事は、後ほど詳しく見ていくことになりますが cum_regret および cum_strategy のうち、手番が player であるような部分の更新を行うことです。逆に言えば、これらのフィールドのうちの手番が player でないような部分は更新が行われません。これは一見するとおかしな実装で、バニラな CFR アルゴリズムはすべてのプレイヤーのリグレットや戦略を**同時更新 (simultaneous update)** することを前提にしていたはずなのに、この実装ではリグレットや戦略がプレイヤー毎に**交互更新 (alternating update)** されることになります。果たしてそれで大丈夫なのかと思われるかもしれませんが、実用上は交互更新の方が収束が早いことが知られており、そのため今回の実装でも交互更新をあえて採用しています。理論的にも、解析は難しくなるものの収束は引き続き保証されるようです[14]。

最後に、compute_average_strategy() を呼び出して各時刻の重み付き平均戦略を返します。compute_average_strategy() も愚直に書き下せばよく、実装は次のようになるでしょう：

```
/// フィールド `cum_strategy` を参照して平均戦略を返す
fn compute_average_strategy(&self) -> HashMap<PublicHistory, Vec<Vec<f64>>> {
    let num_private_hands = T::num_private_hands();
    let mut average_strategy = self.cum_strategy.clone();

    for strategy in average_strategy.values_mut() {
        let mut denom = vec![0.0; num_private_hands];
        strategy.iter().for_each(|strategy_action| {
            add_assign_vec(&mut denom, &strategy_action);
        });

        strategy.iter_mut().for_each(|strategy_action| {
            div_assign_vec(strategy_action, &denom, 0.0);
        });
    }

    average_strategy
}
```

14 N. Burch, M. Moravcik, and M. Schmid. Revisiting CFR+ and alternating updates. Journal of Artificial Intelligence Research, 64:429–443. 2019.

ここで、add_assign_vec といった関数がいくつか出現していますが、これは配列の各要素を加算代入したりする関数で、次のような定義を持ちます。処理の内容は何となく分かると思いますので、いちいち全部紹介はしないことにします。

```
fn add_assign_vec(lhs: &mut Vec<f64>, rhs: &Vec<f64>) {
    lhs.iter_mut().zip(rhs).for_each(|(l, r)| *l += *r);
}
```

Counterfactual value の再帰的な計算

それでは、先ほど後回しにしてしまった cfr_recursive() の実装に移っていきます。先述したように、cfr_recursive() の仕事は cum_regret と cum_strategy の更新ですが、関数そのものの返り値は counterfactual value となっていて、関数名の通りこれを再帰的に計算します。

さすがにこの関数はそれなりの実装量になってしまうため、紙面で読むのはややタフかもしれませんが、頑張って見ていきましょう。

```
/// `player` の counterfactual value を再帰的に計算する
fn cfr_recursive(
    &mut self,
    node: &T::Node,
    player: usize,
    pi: &Vec<f64>,
    pmi: &Vec<f64>,
) -> Vec<f64> {
    // 終端履歴なら単に counterfactual value を返す
    if node.is_terminal() {
        return self.game.evaluate(node, player, pmi);
    }

    // 現在のパブリックな履歴を取得
    let public_history = node.public_history();

    // 現時刻の戦略を regret-matching アルゴリズムによって求める
    let mut strategy = Self::regret_matching(&self.cum_regret[public_history]);
```

```
// 返り値となる counterfactual value を 0 で初期化
let mut cfvalue = vec![0.0; T::num_private_hands()];

// 手番が `player` の場合
if node.current_player() == player {
    let mut cfvalue_action_vec = Vec::with_capacity(node.num_actions());

    // 各アクションに対する counterfactual value を計算する
    for action in node.actions() {
        let pi = mul_vec(&pi, &strategy[action]);
        let mut cfvalue_action =
            self.cfr_recursive(&node.play(action), player, &pi, pmi);
        cfvalue_action_vec.push(cfvalue_action.clone());
        mul_assign_vec(&mut cfvalue_action, &strategy[action]);
        add_assign_vec(&mut cfvalue, &cfvalue_action);
    }

    // リグレットの累積値と戦略の和を更新
    for action in node.actions() {
        let cum_regret: &mut Vec<f64> =
            &mut self.cum_regret.get_mut(public_history).unwrap()[action];
        let cum_strategy: &mut Vec<f64> =
            &mut self.cum_strategy.get_mut(public_history).unwrap()[action];

        cum_regret.iter_mut().for_each(|el| {
            *el *= if *el >= 0.0 {
                self.alpha_t
            } else {
                self.beta_t
            }
        });

        add_assign_vec(cum_regret, &cfvalue_action_vec[action]);
        sub_assign_vec(cum_regret, &cfvalue);

        mul_assign_scalar(&mut strategy[action], self.gamma_t);
```

```
                mul_assign_vec(&mut strategy[action], &pi);
                add_assign_vec(cum_strategy, &strategy[action]);
            }
        }
        // 手番が `player` でない場合
        else {
            for action in node.actions() {
                let pmi = mul_vec(&pmi, &strategy[action]);
                add_assign_vec(
                    &mut cfvalue,
                    &self.cfr_recursive(&node.play(action), player, pi, &pmi),
                );
            }
        }

        cfvalue
    }
```

まず、はじめに関数 `cfr_recursive()` の引数についてですが、`node` と `player` については特に説明は要らないでしょう。`pi` と `pmi` は、現在の引数 `node` への到達確率のうち `player` の寄与と `player` 以外の寄与を表しています。また、返り値は counterfactual value となっています。ここで `pi`, `pmi` および返り値の型が実数の配列となっているのはどういうことかというと、例によってこれらの配列の各要素は `player`（`pmi` については相手のプレイヤー）が特定の手札を持っているときの値を表しています。

この再帰関数の終了条件は、引数 `node` が終端履歴に達することです。終端履歴に達したら、ゲームの定義に基づいて counterfactual value を計算し、その値を返します。引数 `node` が終端履歴でなかった場合、再帰的に子ノードの計算を行うことになります。

それでは引数 `node` が終端履歴でなかった場合の処理についてですが、まず現時刻における戦略が regret-matching アルゴリズムによって求められます。この regret-matching アルゴリズムの実装は簡単ですね：

```
    /// regret-matching アルゴリズム
```

```
fn regret_matching(regrets: &Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    let num_actions = regrets.len();
    let num_private_hands = T::num_private_hands();
    let mut strategy = regrets.clone();

    let mut denom = vec![0.0; num_private_hands];
    strategy.iter_mut().for_each(|strategy_action| {
        nonneg_assign_vec(strategy_action);
        add_assign_vec(&mut denom, strategy_action);
    });

    strategy.iter_mut().for_each(|strategy_action| {
        div_assign_vec(strategy_action, &denom, 1.0 / num_actions as f64);
    });

    strategy
}
```

あとは counterfactual value の計算を再帰的に行いながら、現在の手番が `player` である場合は `cum_regret` と `cum_strategy` も併せて適切に更新していくだけです。…… と、文章で書くのは簡単ですが、足し算や掛け算などをバグ無く正確に記述するのは結構大変な作業です。行数で言ってしまえば 60 行程度とそれほど長いというわけではない関数ですが、デバッグも容易ではありませんし、実装の与えられていない状態から書き起こそうと思うとなかなか骨が折れるでしょう。

以上で CFR アルゴリズムの実装は完成です。ここまで出来てしまえば `Game` トレイトを満たすようにゲームの定義を行ってあげれば CFR アルゴリズムを簡単に適用することができます。次章以降では、ゲームのインスタンスをいくつか定義して実際に CFR アルゴリズムを動かしていきます。

4.3. ユーティリティ関数の実装

と、ゲームの各インスタンスを見に行く前に、得られた戦略を解析するユーティリティ関数もついでに実装してしまいましょう。具体的には、戦略の組が与えられたときに利得の期待値を計算する関数と、可搾取量を計算する関数を実装していきます。

利得の期待値の計算

まず利得の期待値を計算する関数の方から見ていきます。と言っても、再帰関数に抵抗が無い方ならば利得の期待値を計算する処理は素直に書けるでしょう。問題は `Game` トレイトの `evaluate()` メソッドが厄介な型をしているという点ですが、まず求める利得の期待値 $u_i(\sigma)$ は

$$u_i(\sigma) = \sum_{z \in Z} \pi^\sigma(z) \cdot u_i(z)$$

であり、メソッド `evaluate()` は counterfactual value

$$v_i(\sigma, z) = \pi^\sigma_{-i}(z) \cdot u_i(z)$$

を計算してくれるので、あとは $\pi^\sigma(h) = \pi^\sigma_i(h) \cdot \pi^\sigma_{-i}(h)$ であるということを思い出せば、終端履歴における処理の内容も理解できると思います。

```
/// 戦略 `strategy` のもとでの `player` の利得の期待値を返す
pub fn compute_ev<T: Game>(
    game: &T,
    player: usize,
    strategy: &HashMap<PublicHistory, Vec<Vec<f64>>>,
) -> f64 {
    let ones = vec![1.0; T::num_private_hands()];
    compute_ev_rec(game, &T::root(), player, &ones, &ones, strategy)
}

/// 利得の期待値を再帰的に計算するヘルパー
fn compute_ev_rec<T: Game>(
    game: &T,
    node: &T::Node,
    player: usize,
    pi: &Vec<f64>,
    pmi: &Vec<f64>,
    strategy: &HashMap<PublicHistory, Vec<Vec<f64>>>,
) -> f64 {
    if node.is_terminal() {
        // `dot` は内積を計算する関数
        return dot(&game.evaluate(node, player, pmi), &pi);
    }
```

```
    let current_strategy = &strategy[node.public_history()];
    if node.current_player() == player {
        node.actions()
            .map(|action| {
                let pi = mul_vec(&current_strategy[action], &pi);
                compute_ev_rec(game, &node.play(action), player, &pi, pmi, strategy)
            })
            .sum()
    } else {
        node.actions()
            .map(|action| {
                let pmi = mul_vec(&current_strategy[action], &pmi);
                compute_ev_rec(game, &node.play(action), player, pi, &pmi, strategy)
            })
            .sum()
    }
}
```

可搾取量の計算

さて、残った方の可搾取量を計算する関数ですが、こちらはちょっとマジカルな感じもある実装となっています。実装は簡潔だが、なぜこれで正しく動くのかがなかなか分からないという類のやつですね。何はともあれ実装を先に見てみることにしましょう。

```
/// 戦略の組 `strategy` の可搾取量を返す
pub fn compute_exploitability<T: Game>(
    game: &T,
    strategy: &HashMap<PublicHistory, Vec<Vec<f64>>>,
) -> f64 {
    let ones = vec![1.0; T::num_private_hands()];
    let br0 = best_cfvalues_rec(game, &T::root(), 0, &ones, strategy);
    let br1 = best_cfvalues_rec(game, &T::root(), 1, &ones, strategy);
    br0.iter().sum::<f64>() + br1.iter().sum::<f64>()
}
```

```
/// 最適反応戦略の counterfactual value を再帰的に計算するヘルパー
fn best_cfvalues_rec<T: Game>(
    game: &T,
    node: &T::Node,
    player: usize,
    pmi: &Vec<f64>,
    strategy: &HashMap<PublicHistory, Vec<Vec<f64>>>,
) -> Vec<f64> {
    if node.is_terminal() {
        return game.evaluate(node, player, pmi);
    }

    if node.current_player() == player {
        node.actions()
            .map(|action| {
                best_cfvalues_rec(game, &node.play(action), player, pmi, strategy)
            })
            .reduce(|v, w| max_vec(&v, &w))
    } else {
        let current_strategy = &strategy[node.public_history()];
        node.actions()
            .map(|action| {
                let pmi = mul_vec(&pmi, &current_strategy[action]);
                best_cfvalues_rec(game, &node.play(action), player, &pmi, strategy)
            })
            .reduce(|v, w| add_vec(&v, &w))
    }
    .unwrap()
}
```

ヘルパー関数である `best_cfvalues_rec()` は、コメントにも書いてあるように、`player` が最適反応戦略を取ったときの counterfactual value を再帰的に求める関数です。 実際、 終端履歴に到達したら単に `evaluate()` を呼んでいるだけであることが分かります。

さて、実はこの counterfactual value は利得の期待値への寄与とイコールになっているのですが、 ここで 「最適反応戦略には必ず純粋戦略が含まれている」 という重要な事実を利用し

ています。というのも、player 以外の戦略は固定されているものとしているのですから、自分が相手に搾取されてしまう可能性は考える必要がなく、従って単に利得の期待値が最も大きくなるようなアクションを常に選べば利得を最大化できるためです。よって、各履歴 h における $\pi_i^\sigma(h)$ は 0 か 1 のどちらかであるとしてよく、$\pi_i^\sigma(h)$ を追跡したりする必要が無いのです。

よって、best_cfvalues_rec() は $\pi_{-i}^\sigma(h)$ だけを引数にとって追跡します。 現在の node の手番が player であるなら、counterfactual value、すなわち利得の期待値への寄与が最大となるアクションを選択し、手番が player でないなら単に counterfactual value を加算します。

5. CFR の具体例 1: Kuhn Poker

それでは、本章からはいくつかのゲームの定義を実装し、CFR アルゴリズムを実際に動かしていきます。まずは Kuhn poker と呼ばれる非常に単純化されたポーカーの変種を扱っていくことにしましょう。

5.1. Kuhn Poker のルール

Kuhn poker は 2 人向けのポーカーで、デッキは 3 枚のカードのみから構成されます。ここでは、キング (K)、クイーン (Q)、ジャック (J) の 3 枚を用いることとし、先に挙げたカードほど強いものとします。

ここからはゲームの進行について説明していきますが、まず 2 人のプレイヤーは相手から見えないようにカードをデッキから 1 枚ずつ引き、残った 1 枚については伏せたままにしておきます。続いて、各プレイヤーは 1 点の**アンティ**（参加費）を場に供託します。どちらかのプレイヤーが**ベット**することでもう 1 点を追加で供託することもできますが、この手順については後ほど説明します。供託する点数について両者の合意が取れた場合、各プレイヤーは自分の持つカードを公開し、より強いカードを持っていた方がそれまでに供託されていた点数を総取りします（**ショーダウン**）。

アンティを供託した後は、プレイヤーは先手と後手に分かれて、次のような手順を経て供託する点数についての合意を取ろうとします：

- 先手は手番をパスする（**チェック**）か、1 点を追加で供託する（**ベット**）かを選択します。
  - 先手がチェックした場合、後手は同様に手番をパスしてショーダウンに進む（チェック）か、1 点を追加で供託する（ベット）かを選択します。

- 後手がベットした場合、先手はすでに供託した１点を相手に与えることにして勝負から降りる（**フォールド**）か、後手と同様に１点を追加で供託してショーダウンに進む（**コール**）かを選択します。
- 先手がベットした場合、後手はすでに供託した１点を相手に与えることにして勝負から降りる（フォールド）か、先手と同様に１点を追加で供託してショーダウンに進む（コール）かを選択します。

以上が Kuhn poker のルールですが、非常に単純化されたポーカーであるとは述べたものの、最適な戦略がどのようなものであるかを思いつくのはそう簡単ではないでしょう。 キングを持っていて相手にベットされた場合は常にコールする、またジャックを持っていて相手にベットされた場合は常にフォールドする、さらに後手がキングを持っていて先手がチェックした場合は常にベットする、くらいのことは分かるでしょうが、それ以上のことは少なくとも筆者にはまったく明らかではありません。

この Kuhn poker は発案者の Harold W. Kuhn によって解析もなされており、ナッシュ均衡が数学的に求まっています。それぞれの具体的な最適戦略は次に挙げる通りですが、ここで先手は最適戦略を無限に持っており、パラメータ $\alpha \in [0, 1/3]$ を自由に選択することができます：

【先手】

- 手札がキング : 確率 3α でベットし、自分のチェックに対して後手がベットした場合は常にコール
- 手札がクイーン : 常にチェックし、後手がベットした場合は $\alpha + 1/3$ の確率でコール
- 手札がジャック : 確率 α でベットし、自分のチェックに対して後手がベットした場合は常にフォールド

【後手】

- 手札がキング : 常にベットかコール
- 手札がクイーン : 先手がチェックした場合はチェック、先手がベットした場合は確率 $1/3$ でコール
- 手札がジャック : 先手がチェックした場合は確率 $1/3$ でベット、先手がベットした場合は常にフォールド

なお、両プレイヤーがこのナッシュ均衡に基づく戦略を取った場合、後手が平均して 1/18 点だけ勝つことが知られています。

この最適戦略を眺めて個人的に面白いと思うのは、先手はパラメータ α の値にもよりますが、両プレイヤーとも最弱のカードであるジャックを持っているときにも一定の確率でベットを行い、ブラフを仕掛けるのが最適とされている点です。特にポーカーの楽しさの大部分は相手がブラフをしているかどうかの心理的な読み合いにあると思うのですが、適切な確率でブラフを仕掛けることはナッシュ均衡の観点からもちゃんと正当化されるのだということが、このように非常に単純化されたルールのもとでも確認することができます。

5.2. ゲーム定義の実装

Kuhn poker のルールについての説明が済んだので、 続いてゲーム定義の実装に移っていきましょう。 まず、Kuhn poker には特に設定を変更できる部分がありませんので、 構造体 `KuhnGame` には空の定義を持たせます。`KuhnGame` のノードを表す型である `KuhnNode` はパブリックな履歴を保持すれば良いので、唯一のフィールド `public_history` を持ちます。また、ノードはクローンできると便利なので、`KuhnNode` は `Clone` トレイトを継承しておきます。

```
pub struct KuhnGame {}

#[derive(Clone)]
pub struct KuhnNode {
    public_history: PublicHistory,
}
```

それでは `KuhnGame` の実装に移っていきますが、`Node`, `root()`, `num_private_hands()` の実装は明らかで、特に考える必要はないでしょう。

```
impl Game for KuhnGame {
    type Node = KuhnNode;

    fn root() -> KuhnNode {
        KuhnNode {
            public_history: Vec::new(),
```

```
        }
    }

    fn num_private_hands() -> usize {
        3
    }

    fn evaluate(&self, node: &KuhnNode, player: usize, pmi: &Vec<f64>) -> Vec<f64> {
        ...
    }
}
```

ここで、やはり問題となるのが `evaluate()` の実装です。`evaluate()` がどのようなメソッドだったかを思い出すと、「終端履歴 `node` において、最初の偶然手番の寄与を含まない counterfactual 到達確率が `pmi` のときの `player` の counterfactual value を計算する」というものでした。また、counterfactual value は終端履歴においては counterfactual 到達確率と利得の積で与えられるのでしたね。

一つ注意したいのは、返り値となる counterfactual value の計算においては最初の偶然手番による寄与も含む counterfactual 到達確率の値を用いるべきですが、引数 `pmi` には最初の偶然手番による寄与が含まれていないという点です。つまり、最初の偶然手番によってどのような確率でどのようなハンドが各プレイヤーに与えられるのかは、この `evaluate()` メソッド内で定義を与えなければならないということです。`evaluate()` を `GameNode` ではなく `Game` トレイトのメソッドとしている理由はこのためでもありますが、今回の Kuhn poker の例では両プレイヤーへのカードの配り方は全部で 6 通りあり、それらが等確率に発生するということをこの `evaluate()` メソッドに組み込む必要があります。

以上の考察を経ると、利得を計算する関数 `payoff()` を用いて `evaluate()` が次のように書き下せることが分かるでしょう：

```
fn evaluate(&self, node: &KuhnNode, player: usize, pmi: &Vec<f64>) -> Vec<f64> {
    let mut cfvalue = vec![0.0; Self::num_private_hands()];

    for my_card in 0..Self::num_private_hands() {
```

```
        for opp_card in 0..Self::num_private_hands() {
            if my_card == opp_card {
                continue;
            }
            cfvalue[my_card] +=
                Self::payoff(node, player, my_card, opp_card) * pmi[opp_card] / 6.0;
        }
    }

    cfvalue
}
```

さらに、以下の実装を加えて構造体 `KuhnGame` の定義を完成させます：

```
const CHECK_FOLD: usize = 0;
const BET_CALL: usize = 1;

impl KuhnGame {
    /// コンストラクタ
    pub fn new() -> Self {
        Self {}
    }

    fn payoff(
        node: &KuhnNode,
        player: usize,
        my_card: usize,
        opp_card: usize,
    ) -> f64 {
        match (node.public_history.as_slice(), node.public_history.last()) {
            ([CHECK_FOLD, CHECK_FOLD], _) if my_card > opp_card => 1.0,
            ([CHECK_FOLD, CHECK_FOLD], _) => -1.0,
            (_, Some(&CHECK_FOLD)) if node.current_player() == player => 1.0,
            (_, Some(&CHECK_FOLD)) => -1.0,
            _ if my_card > opp_card => 2.0,
            _ => -2.0,
```

```
        }
    }
}
```

最後に、ゲーム木のノードを表す型である KuhnNode の実装を以下のように与えます。こちらも特に難しい点は無く、素直に定義を書き下しているだけであることが分かると思います。

```
impl GameNode for KuhnNode {
    fn public_history(&self) -> &PublicHistory {
        &self.public_history
    }

    fn is_terminal(&self) -> bool {
        match self.public_history.as_slice() {
            [CHECK_FOLD, BET_CALL] => false,
            [_, _] => true,
            [_, _, _] => true,
            _ => false,
        }
    }

    fn current_player(&self) -> usize {
        self.public_history.len() % 2
    }

    fn num_actions(&self) -> usize {
        2
    }

    fn play(&self, action: Action) -> Self {
        let mut ret = self.clone();
        ret.public_history.push(action);
        ret
    }
}
```

5.3. プログラムの実行

ここまで実装パートが長かったですが、いよいよプログラムを実際に実行していきましょう。`main()` 関数を次のように記述することで、これまでに作ってきたコンポーネントや関数を簡単に組み合わせて実行することができます：

```
fn main() {
    // CFR の反復回数を指定
    let num_iterations = 10000;

    // Kuhn poker のゲーム定義のインスタンスを作成
    let kuhn_game = KuhnGame::new();

    // CFR を管理する構造体のインスタンスを作成
    let mut cfr = CFRMinimizer::new(&kuhn_game);

    // CFR を実行して最適戦略を得る
    let strategy = cfr.compute(num_iterations);

    // 利得の期待値と可搾取量を計算
    let ev = compute_ev(&kuhn_game, 0, &strategy);
    let exploitability = compute_exploitability(&kuhn_game, &strategy);

    ...
```

さて、今回は CFR の反復回数を 10000 回に指定しましたが、これで正しい結果はちゃんと得られるのでしょうか。出力を適当に整形すると、手元のマシンでは 0.1 秒も掛からずに次のような出力が得られました：

```
[First player]
- EV: -0.0556
- Bet%                          - Call% (Check => Bet => ?)
    K: 56.72%                       K: 100.00%
    Q: 0.00%                        Q: 52.25%
    J: 18.90%                       J: 0.00%
```

```
[Second player]
- EV: +0.0556
- Bet% (Check => ?)             - Call% (Bet => ?)
    K: 100.00%                     K: 100.00%
    Q: 0.00%                       Q: 33.33%
    J: 33.33%                      J: 0.00%
```

```
- Exploitability: +4.774e-5
```

まず、可搾取量は 4.8×10^{-5} 未満となっており、実用上は十分収束していることが分かります。これだけ収束していれば、得られた戦略もほぼナッシュ均衡となっているはずで、数学的に求まっている最適戦略と実際に比較してみると $\alpha \approx 18.91\%$ のときの最適戦略とほぼ一致していることが見て取れます。先手および後手の利得の期待値もちゃんと $\mp 1/18$ になっていますね。

ここまで理論も実装もそれなりに重たかったかと思いますが、実際にプログラムが動いたときの喜びもその分大きいものです。完全情報ゲームであれば、例えば三目並べのような比較的単純なゲームの最適戦略を求めたければ単なる深さ優先探索をするだけですが、 不完全情報ゲームは、この Kuhn poker のように例え非常に単純化されたルールのものであっても、プログラムによって最適戦略を求めようと思うとこれだけの準備が必要となってしまうのです。逆に言えば、不完全情報ゲームの魅力や奥の深さはこのような性質によるところが大きいとも言えるでしょう。

6. CFR の具体例 2: プッシュ / フォールド

さて、前章で見てきた Kuhn poker はルールが非常に単純化されているもので、ナッシュ均衡解が解析的に求まっているような言わばおもちゃに過ぎませんでした。おもちゃであってもきちんと解析を行うことは大事なことではありますが、具体例がそれだけでは物足りなさもあるでしょう。そこで本章では、実際のテキサスホールデムにおいてしばしば発生する単純な状況をモデル化した**プッシュ / フォールド**ゲームについて解析を行っていきます。

6.1. プッシュ / フォールドのルール

本章で扱うプッシュ / フォールドゲームは、 テキサスホールデムの用語が分かる方には「**ス**

モールブラインドが**プリフロップ**で**オールイン**か**フォールド**しかしない**ヘッズアップ**のテキサスホールデム」で通じるのですが、そうでない方には暗号か何かにしか見えないでしょう。そういったテキサスホールデムに馴染みが無い方に向けて、ここからは最低限のルールの説明をしていきます。

まず、**ヘッズアップ**というのは一対一、すなわち二人で行われるポーカーゲームのことを指します。ヘッズアップでは、プレイヤーは**スモールブラインド**と**ビッグブラインド**に分かれます。ここで、両者とも点数を強制的に供託させられますが、スモールブラインドが供託しなければならない点数はビッグブラインドの半分となっています。ビッグブラインドが強制的に供託させられる点数は bb という単位として扱われます。

続いて、テキサスホールデムでは両プレイヤーに二枚の手札が配られますが、この手札が配られた直後のラウンドを**プリフロップ**と呼びます。プリフロップでは、スモールブラインドが先にアクションを起こします。ここで、プッシュ / フォールドでは手持ちの点数をすべて供託する（**オールイン**）か、すでに供託した 0.5[bb] を相手に与えることにして勝負から降りる（**フォールド**）かのどちらかのアクションのみを取ることができます。スモールブラインドがオールインした場合、ビッグブラインドは供託する点数を相手に合わせて勝負に乗る（**コール**）か、すでに供託した 1[bb] を相手に与えることにして勝負から降りる（フォールド）かを選択することになります。

オールインに対してビッグブラインドがコールした場合は**ショーダウン**へと進むことになり、より強い役を作った方がそれまでに供託された点数を総取りします。テキサスホールデムでは二枚の手札に加えて、コミュニティカードと呼ばれる全員が使用できるカードが五枚公開され、これらの計七枚のうちもっとも強い五枚の組み合わせを用いて役の強さを競います。

役については、多くの方に馴染みのあるであろうポーカーと同じなので特に説明はしませんが、同一の役であってもカードの**ランク**によってタイブレークがなされる点は注意が必要かもしれません。例えば、エースのワンペアはキングのワンペアよりも強く、また同じエースのワンペアどうしであっても持っているカードのうち次に強いカードがキングの場合はそれがクイーンの場合よりも強いものとして扱われます。なお、スペードやハートといったカードの**スート**は役の強さに影響しません。

さて、ルールの最低限の説明は以上になりますが、なぜこのようなゲームを考えているのかについても簡単に説明することにしましょう。通常のテキサスホールデムではスモールブライ

ンドはオールインとフォールドの他にも、供託する点数を合わせる**コール**や、オールインまでは行かないが供託する点数を増やす**レイズ**というアクションが行えます。このコールやレイズといったアクションを除外してしまったモデルに意味があるのかというと、両プレイヤーの持ち点の最小値（**エフェクティブスタック**）が小さい場合は、選択肢をオールインとフォールドのみに限ってもそれなりに良い戦略となることが知られています。というのも、中途半端なコールやレイズはビッグブラインド側のオールインを簡単に誘発してしまうためで、それならば最初から選択肢をオールインかフォールドに限ってしまうのは合理的と言えます[*15]。

6.2. ゲーム定義の実装

ルールについての説明が済んだので、ゲーム定義の実装の説明に移っていきましょう。なお Kuhn poker の例では実装をすべて紹介しましたが、今回は実装の紹介は省略します。

ここまで読んでこられた方には予想がついているかと思いますが、ゲーム定義の実装の鬼門はやはり `evaluate()` メソッドです。今回は手札が二枚あるため `num_private_hands()` の値は ${}_{52}\mathrm{C}_2 = 1{,}326$ となっており、相手も同じく二枚の手札を持っていますから、ショーダウンまで進んだ場合は ${}_{52}\mathrm{C}_2 \times {}_{50}\mathrm{C}_2 = 1{,}624{,}350$ 個のパターンの計算を `evaluate()` で行う必要があります。

この時点ですでにやや暗雲が立ち込めている気がしますが、さらに問題となるのはショーダウン時の勝率を計算しようと思うと ${}_{48}\mathrm{C}_5 = 1{,}712{,}304$ 通りのコミュニティカードの組み合わせを総当たりする必要が生じることです。${}_{52}\mathrm{C}_2 \times {}_{50}\mathrm{C}_2 \times {}_{48}\mathrm{C}_5 \approx 2.8 \times 10^{12}$ ですから、このオーダーの計算を繰り返すのはまったく現実的ではありません。

ただ幸いなことに、自分と相手の手札が与えられたときの勝率は一定ですので、今回の状況ではそれを**前計算**してしまうことが可能です。およそ 2.8×10^{12} 通りの総当たりはやはり必要になってしまいますが、一度その計算さえできてしまえば結果を使い回すことができます。前号の vol.6 で紹介した高速役判定プログラムを用いることで、筆者の 16 スレッドのマシンでは 20 分掛からず前計算を完了できました（自画自賛ですがこれは非常に高速だと思います）。

あとは、自分の手札と相手の手札でカードが重複しないように気をつけながら、それぞれの場合の counterfactual value を計算する処理を書き下せば良いことになります。これも文章で

15 エフェクティブスタックが小さい状況では中途半端なレイズが悪手なのは間違いないのですが、最近の解析ではコールは実は強力なことが分かりつつあります。ただし、コールを戦略に組み込む場合は相手のオールインに対するディフェンスや、プリフロップ以降の戦略についても考えなければなりません。よって、コールは上級者に向いたアクションであると言え、初心者にはやはりオールインかフォールドに限った戦略の方が扱いやすいでしょう。

書くのは簡単だが実際に実装するのはハマると大変という類のものですが、今回の場合は完成した実装がありますので、気になる方は４章で紹介した URL からリポジトリを参照ください。

ところで、手札の組み合わせは ${}_{52}C_2 = 1,326$ 通りであると述べましたが、これを 169 通りの同値類に潰せるのではないかと思われた賢い方もいるかもしれません。というのも、例えばA♠A♥のペアとA♦A♣のペアは異なる手札ですが、勝率の観点からは同じであるため、これらをまとめて扱うことが一見可能なように思えるためです。

しかし実際にはこれは横着というもので、相手の手札を特定のものに固定して考えると勝率は異なるため、このようなことを行うとプログラムが上手く動かなくなってしまいます。例えば相手の手札をK♠K♥に固定したとすると、A♠A♥のペアはフラッシュで負ける可能性が無いのに対し、A♦A♣のペアはスペードかハートのフラッシュを相手に作られると負け得るため、勝率が異なることが分かります。従って、`evaluate()` では ${}_{52}C_2 = 1,326$ 通りのすべての組み合わせをちゃんと試す必要があるのです[16]。

6.3. プログラムの実行

さて、それではゲーム定義が実装できたものとしてプログラムを実際に実行していきましょう。エフェクティブスタックは 10[bb]、反復回数は 1000 回に指定することにすると、`main()` 関数は次のようになります：

```
fn main() {
    // エフェクティブスタックを指定
    let effective_stack = 10.0;

    // CFR の反復回数を指定
    let num_iterations = 1000;

    // プッシュ / フォールドゲームの定義のインスタンスを作成
    let push_fold_game = PushFoldGame::new(effective_stack);

    // CFR を管理する構造体のインスタンスを作製
    let mut cfr = CFRMinimizer::new(&push_fold_game);
```

16 計算結果が厳密でなくなることを承知の上で、あえてこういった「横着」をするという方針もあり得ます。そのようなテクニックは**抽象化**と呼ばれ、最先端のソルバーなどは多かれ少なかれ何らかの抽象化を行っています。

```
    // CFR を実行して最適戦略を得る
    let strategy = cfr.compute(num_iterations);

    // 利得の期待値と可搾取量を計算
    let ev = compute_ev(&push_fold_game, 0, &strategy);
    let exploitability = compute_exploitability(&push_fold_game, &strategy);

    ...
```

手元のマシンでは実行に 10 秒程度を要しましたが、まずは可搾取量を見てみましょう：

```
- Exploitability: +1.447e-7[bb]
```

可搾取量は 1.5×10^{-7} [bb] 未満となっており、これなら十分収束していると言えそうですね。続いてスモールブラインドの最適戦略はどうでしょうか：

```
[Pusher (Small blind)]
- EV: -0.0454[bb]
- Overall push rate: 58.29%
 |   A     K     Q     J     T     9     8     7     6     5     4     3     2
-+-------------------------------------------------------------------------------
A| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%
K| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%
Q| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%
J| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%   -
T| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%   -     -
9| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%   -     -     -
8| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%   -     -
7| 100.% 100.% 100.%   -     -   100.% 100.% 100.% 100.% 100.% 100.%   -     -
6| 100.% 100.%   -     -     -     -     -   100.% 100.% 100.% 100.%   -     -
5| 100.% 100.%   -     -     -     -     -     -     -   100.% 100.% 100.%   -
4| 100.% 100.%   -     -     -     -     -     -     -     -   100.% 71.6%   -
3| 100.% 100.%   -     -     -     -     -     -     -     -     -   100.%   -
2| 100.% 100.%   -     -     -     -     -     -     -     -     -     -   100.%
```

まず、スモールブラインドの利得の期待値は -0.0454[bb] となっており、このゲームはスモールブラインドが若干不利であることが分かります。続いて、全体の 58.29% の手札でオールインをするのが良いと言っており、具体的にどの手札でオールインすべきかが続いて表形式で示されています。この表の見方ですが、対角線の右上は手札のスートが揃っている場合を、対角線の左下は手札のスートが揃っていない場合を表しています。確率的にオールインを選択すべき場面はスートが揃っている 4 と 3 で手札が構成されているときのみで、この場合は 71.6% の確率でオールインを選択すべきという結果になっています。

続いてビッグブラインドの最適戦略についても見ていきましょう：

```
[Caller (Big blind)]
- EV = +0.0454[bb]
- Overall call rate: 37.38%
 |   A     K     Q     J     T     9     8     7     6     5     4     3     2
-+-------------------------------------------------------------------------------
A| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%
K| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.%
Q| 100.% 100.% 100.% 100.% 100.% 100.% 100.% 100.% 40.0%   -     -     -     -
J| 100.% 100.% 100.% 100.% 100.% 100.% 100.%   -     -     -     -     -     -
T| 100.% 100.% 100.% 100.% 100.% 100.%   -     -     -     -     -     -     -
9| 100.% 100.% 100.%   -     -   100.%   -     -     -     -     -     -     -
8| 100.% 100.%   -     -     -     -   100.%   -     -     -     -     -     -
7| 100.% 100.%   -     -     -     -     -   100.%   -     -     -     -     -
6| 100.% 100.%   -     -     -     -     -     -   100.%   -     -     -     -
5| 100.% 100.%   -     -     -     -     -     -     -   100.%   -     -     -
4| 100.%   -     -     -     -     -     -     -     -     -   100.%   -     -
3| 100.%   -     -     -     -     -     -     -     -     -     -   100.%   -
2| 100.%   -     -     -     -     -     -     -     -     -     -     -   100.%
```

スモールブラインド側が上位 58.29% の手札でオールインしてきているので、それに対抗するビッグブラインド側はそれよりタイトな上位 37.38% の手札でのみコールします。コールすべきかどうかの境界となっているのはスートが揃っている Q と 6 で手札が構成されている場合で、このとき 40.0% の確率でコールを選択すべきという結果になっています。

以上がプッシュ / フォールドゲームの解析になりますが、実はこの内容が行えるプログラムに

ユーザーインターフェースを整えた程度のものが世間では有償で提供されていたりもします。説明は割とあっさり済ませてしまいましたが、実装が公開されていてかつきちんと高速に動作するプログラムというのは案外貴重だったりするのではないでしょうか。

7. まとめと参考文献

以前に CFR アルゴリズムを用いた趣味プログラミングを書いたことがあったので、「せっかくだし記事にしてみるか〜」くらいのノリで書き始めたら 40 ページを超える内容となってしまいました。記事にするからには不正確なことは書けませんし、これまで適当に理解したつもりだった内容も論文にちゃんとあたって勉強する必要がたびたび生じ、筆者としても学びの多いプロセスとなりました。

さて本記事はポーカーを念頭に置いたものでしたが、 具体例として出てきたのは非常に限られたゲームであって、 テキサスホールデムの解析には程遠いものでもありました。 実際には 2017 年のほぼ同時期に DeepStack と Libratus という二つのプログラムがヘッズアップのゲームにおいてプロを破っているなど、プログラムの開発者は何らかの方法でテキサスホールデムの複雑さに向き合っていることが分かります（使われている計算資源は一般人からするとしばしば膨大すぎるものではありますが）。

そこで、ポーカー AI の最新の動向を知りたいという方のために DeepStack と Libratus およびそれ以降の代表的なポーカー AI を独断でチョイスして、その名前と対応する論文のリストを時系列順に紹介しようと思います。開発者（研究者）がテキサスホールデムなどの非常に複雑なゲームに対してどう向き合っているのか、気になる方はこれらの論文にぜひあたってみてください。CFR アルゴリズムの基礎を本記事で学んだ後ならば、これらの論文もだいぶ読みやすくなっていることでしょう。

- DeepStack[*17]: ヘッズアップテキサスホールデムで初めてプロを破ったという内容の論文です。 適切にある種の値を管理することで、 ゲーム途中においてこれまでの戦略を忘れても正しく部分ゲームの CFR が再計算できるという continual re-solving と、 探索の深さが一定以上になったら counterfactual value の計算をディープラーニングで学習させた deep CFVnet の推測値で代用するというテクニックを用いています。ターンにおける学習には 175 CPU コア年の計算資源を用いるなど、使用されている計算資源は膨大です。

17 M. Moravčík et al. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. Science, 356 (6337):508–513. 2017.

- **Libratus**[*18]: こちらはヘッズアップテキサスホールデムで「トップ」プロを破ったという内容の論文です。 抽象化されたゲームに対する戦略を前計算しておいて、実行時には抽象度を下げて CFR を再計算するというもので、こちらも必要な計算資源は膨大なものでした。
- **Modicum**[*19]: 非常にコンパクトな計算資源でそこそこ強いポーカー AI を作るという内容の論文です。 具体的には前計算に 700 CPU コア時間と 16GB メモリのみを用い、実行時も 4 コア CPU で平均 20 秒程度の思考時間に収まります。用いられているアルゴリズムは、まずモンテカルロ CFR によって **blueprint** と呼ばれる戦略を前計算し、実行時はその blueprint をもとにさらに探索を行うというものです。この実行時の探索というのが肝なのですが、深さが一定以上になった場合は相手の戦略をいくつかに限定してしまいます。
- **Pluribus**[*20]: これまでのポーカー AI はヘッズアップに限られたものでしたが、 こちらは六人テーブルのテキサスホールデムにおいてトッププロを破ったという内容になっています。使われているアルゴリズムは Modicum と類似しており、blueprint 戦略を利用するというものです。前計算に用いた計算資源は 64 コア CPU を 8 日と 512GB メモリと中程度です。
- **Supremus**[*21]: NeurIPS に投稿したが reject されたのであろう論文です。 ただし論文中で実装されているプログラムは現時点で恐らく最強のもので、DeepStack で用いられていた CFVnet をいろいろ改良した上で計算資源で殴ってみた、という内容になっています。
- **ReBeL**[*22]: 不完全情報ゲームの解釈を変更して「戦略を宣言するとレフェリーがアクションを決定する」ものとして表現すると（**belief representation**）、AlphaZero に代表される「強化学習 + 探索」の枠組みが二人ゼロサムであるような不完全情報ゲームにも適用できるようになるという論文です。ヘッズアップテキサスホールデムでは Libratus と同様にトッププロを有意に破る実力があり、**Liar's Dice** というゲームに対しては実装も公開されています。また、 ポーカーに対する実装では既存のプログラムよりも使われているドメイン知識が少なく、情報の抽象化などをあえて行っていない点も特徴です。

---

18 N. Brown and T. Sandholm. **Superhuman AI for heads-up no-limit poker: Libratus beats top professionals.** Science, 359(6374):418–424. 2018.

19 N. Brown, T. Sandholm, and B. Amos. **Depth-limited solving for imperfect-information games.** In NeurIPS. 2018.

20 N. Brown and T. Sandholm. **Superhuman AI for multiplayer poker.** Science, 365(6456):885–890. 2019.

21 R. Zarick, B. Pellegrino, N. Brown, and C. Banister. **Unlocking the potential of deep counterfactual value networks.** arXiv preprint arXiv:2007.10442. 2020.

22 N. Brown, A. Bakhtin, A. Lerer, and Q. Gong. **Combining deep reinforcement learning and search for imperfect-information games.** In NeurIPS. 2020.

- **Player of Games (PoG)**[*23]: 記事を書いている途中で見つけた DeepMind 社の最新論文です。完全情報ゲームと不完全情報ゲームの両方の探索が統一的に扱える **growing-tree CFR** と呼ばれるアルゴリズムを提案し、完全情報ゲームであるチェスと囲碁、不完全情報ゲームであるヘッズアップテキサスホールデムとスコットランドヤードという四つのゲームにおいて、state-of-the-art とまではいかないが強力な結果を残したという論文です。

さて、長かった本記事もここまでで本当に終わりです。不完全情報ゲームの解析において今や基礎とも言える CFR アルゴリズムについて、背景、数式、実装、具体例を網羅した日本語記事というのも我ながら貴重なのではないかと思うのですが、いかがでしたでしょうか。書く方もそれなりに大変でしたが、ここまでちゃんと読まれた方も大変だったのではないかと思います。この記事が読者の方々に何らかの刺激を与えるものとなっていれば嬉しいですね。それではまた機会がありましたら紙面なりでお会いしましょう。

23 M. Schmid et al. **Player of Games**. arXiv preprint arXiv:2112.03178. 2021.

炒め物のイディオム

zeptometer

注意：この記事では料理の話をします。料理の話をするってことはプログラミングとかそういう話はしないってことなんだ。いいね？

1. はじめに

コロナが新しい生活様式をもたらしてもう 2 年になろうとしています。読者の皆様にも自宅にいる割合が増えたり外食に行く頻度が減った人は多いのではないでしょうか。そうすると選択肢としてはテイクアウト、出前、自炊あたりになってきます。自炊が経済的なのはあたりまえなのですが、いかんせん面倒くさいのがいけない。何を食べるか決めてレシピを見て必要なものを買って実際に料理をする、なんて手間を平日にやるのは結構な苦労です。

特に自炊を普段しない人にとって自炊の手間は大きいものです。例えば手元の食材で何か料理を作ろうにも、その食材でぴったり作れるレシピとなると選択肢が限られてしまいます。あるいはスーパーに買い物に出掛けている時、何か珍しい食材が売っていたり特売で安くなっている食材がある時に、それで何を作れるかその場でスマホでレシピを探すという経験をした方は結構いるのではないでしょうか。

寄稿者はこういう面倒くささはレシピに頼って料理をすることの弊害だと考えています。「どういう料理を作るか」の単位で自炊を組み立てるとあらかじめ何を買うか決めないといけないし手持ちの食材やそスーパーの状況に柔軟に対応することが難しくなります。そうではなくて「それぞれの食材をどう調理するか」の単位で自炊を組み立てることができれば、食材を揃える部分と実際に調理する部分を別々に考えることができるようになって買い物も調理もぐっと楽になります。

この記事ではこのレシピよりも一つ小さい単位である「それぞれの食材をどう調理するか」

の知識を「イディオム」[*24]と呼ぶことにし、イディオムについて解説していきます。紙面と寄稿者の知識の都合もありますので、この記事では炒め物を作る上でのイディオムに限定しておきます。そういうわけで「炒め物のイディオム」になったわけです。

炒め物は比較的シンプルな料理です。おおまかな方針と各食材のイディオムさえおさえておけば、とりあえず今ある食材でそれなりにおいしい炒め物を作ることは難しくありません。組み合わせ次第で色々な料理を作れるので長続きしやすいのも炒め物の利点です。スーパーにおいてあった珍しい食材をとりあえず買っていい感じに炒め物にできた日にはきっと自炊が楽しくなりますよ。

2. 基本の流れ

イディオムに入る前にまず炒め物の基本的な流れを説明します。といっても本当に簡単で、

- 食材は別々に炒めて、最後に弱火で炒めながらまぜる

これだけです。例えば卵ときくらげと豚肉のオイスターソース炒めだと図1のような流れになります[*25]。別々の枝でそれぞれの食材を調理して、最後にそれらを一緒くたにするという過程は炒め物共通です。何故こうするべきかというと、単に食材ごとに適切な加熱具合が異なるからですね。また卵のように別に炒めておいた方が食感が断然よくなるケースもあります。

これを聞くと結構面倒だなと思われるかもしれませんが、これさえやっておけばそれなりにおいしい炒め物ができます。個別においしいレシピを覚えるよりも、「とりあえず別々に炒めておけ」と覚えておいて脳死で手を動かす方が結局楽だと思うんですよね。ちなみに炒めた後の食材は別に同じ皿に入れてしまっても大丈夫です。

味付けをどこでやるかはケースバイケースですが、基本的には食材の一つを炒める際に味をつけるか、あるいは最後に食材を全部混ぜてから味をつける感じになります。詳しいことは次に説明します。

---

24 これは言語のイディオムというよりはプログラミングにおけるイディオムからとっています。「こういう場面ではこういうコードの書き方をすると便利」という知識ですね。イディオムだけではプログラムにはならないがそれを組み合わせることで良いプログラムができる、という部分を料理と対比しています。(これがやりたかっただけ)

25 この「概念図」は小林銅蟲「めしにしましょう」から拝借しています。この記法を使うと依存関係がわかりやすいので全てのレシピ本に導入してほしい

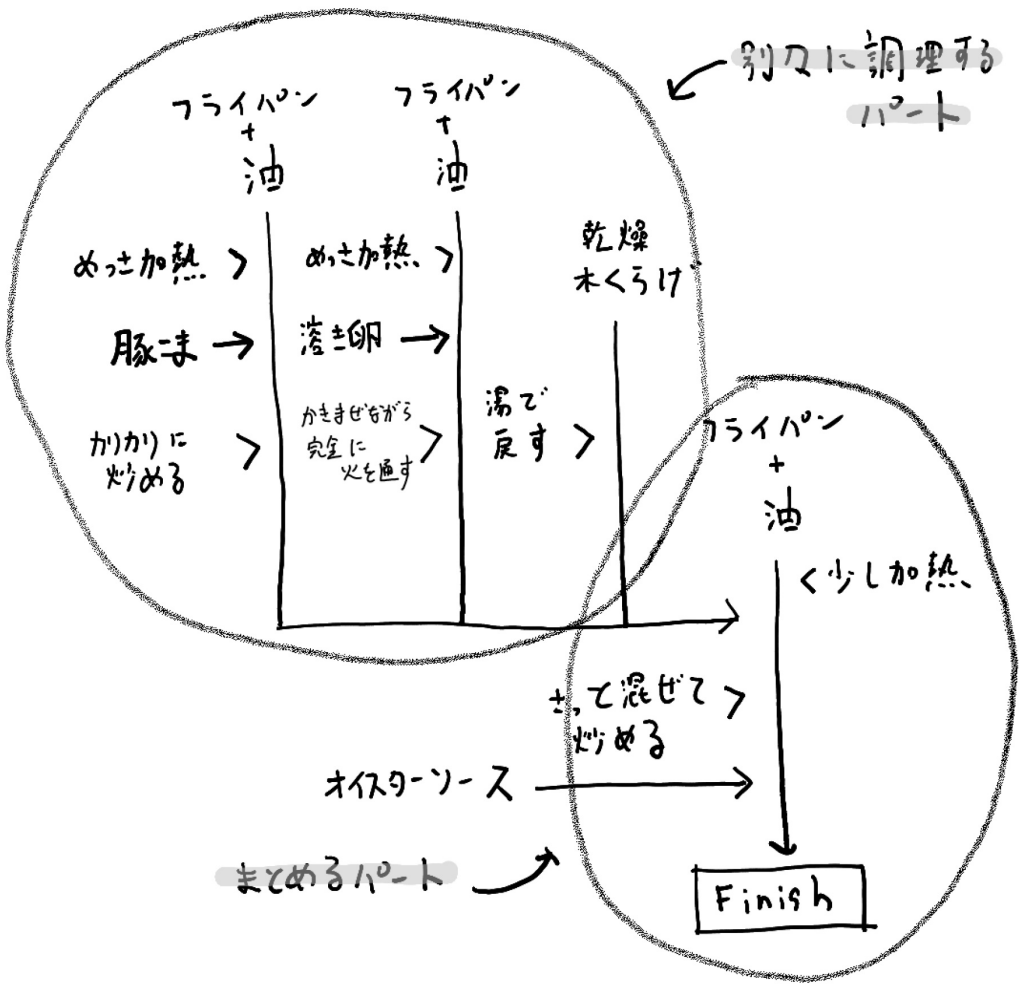


図１　豚と卵と木くらげのオイスターソース炒めの概念図

3. 炒め物の味付け

食材の個別のイディオムについては後ほど説明するとして、全体の味付けをどうするかも大事な要素です。ここでは簡単にできる味付けをいくつか紹介します。

3.1. 油は基本

炒め物には油を入れましょう。テフロンのフライパンでも関係ありません。サラダ油で問題ありませんが、ごま油やオリーブオイルを使うと香りづけになるのでそれもありです。脂身多めの肉を使う場合はそれをあらかじめ炒めて、出てきた油を使うという方法もあります。量は適当でいいのですが、加減がわからない人は一人前で大さじ 1,2 杯程度で試してみるといいでしょう。何回か炒め物を作っているうちに自分にとって丁度いい量がわかってくるはずです。

3.2. 味付けは一種類で OK

料理レシピを見てると「醤油とみりんと日本酒を各小さじ 1」みたいに色々な調味料で味つけしていることが多いです。これは面倒くさいのでやめちゃいましょう。味付けに使う調味料は 1 種類でいいです、そうしちゃえば各調味料のバランスを考えずに量だけ調節すればよくなるので楽です。おすすめの調味料は以下の通りです。

- 塩：塩だけでも全然大丈夫です。 素材の味を活かせるしね。 好きなタイミングで加えましょう
- 醤油：和風の味付け。好きなタイミングでぶっかけましょう。
- オイスターソース：食材を混ぜて最後にオイスターソースを加えるとうまみたっぷりの炒め物ができます。李錦記のオイスターソースは結構いい値段しますが、業務用スーパーに行けば 1 本 100 円くらいの安いオイスターソースを買えます。それくらいのやつでも普通に美味しいです。

3.3. 味見をしろ

この記事では調味料の量とかは書いていません。食材の種類や量によって適切な量が変わるのでそもそも書けないんですね。なので味付けをする際は最初は少なめに入れて、味見をしつつ必要なら追加していくようにしましょう。 マズいメシの原因の 90% は味見をしないことらしいんだ[*26]、これはめちゃくちゃ大事だぞ。

最初の量もわからんという方もいると思うので、この量なら安全であろうという一人前の調味料の量を置いておきます。

---

26 ソースはない。ちなみに残りの 10% はそもそも味覚がぶっ壊れているケースで、この場合は仕方ないから塩分をとりすぎないようにだけ注意しような。

- 塩：一つまみ
- 醤油：小さじ 1
- オイスターソース：大さじ 1

4. 炒め物のイディオム

それではいよいよイディオムの話をしていきましょう。

4.1. 葉野菜

例：小松菜、レタス、白菜、（葉野菜じゃないけど）ピーマン

基本的に葉野菜は思いきり高火力で炒めればなんとかなります。野菜炒めは単体でも十分美味しいですよ。特にレタスはシャキシャキした食感がよいのでおすすめです。詳しい手順を見ていきましょう。

1. 水につけておく

葉野菜は炒める前にしばらく水につけておくといいです。 時間がなかったら焼く前に水でざっと洗うだけでも大分違います。野菜は生きているので水をやると吸収してしゃきっとします。ついた水は完全に落とさなくても大丈夫で、むしろ野菜を蒸し焼きにしつつ焦げつくのを防げるので好ましいです。

2. 油とフライパンを加熱する

フライパンに油を入れて強火を入れます。中華鍋だったら煙が出るくらいにガンガンに加熱しちゃっていいんですが、テフロンだと痛んでしまうのでほどほどにしましょう。

3. 炒める

フライパンを十分に加熱できたら切った野菜を入れましょう。ジューッとすごい音がするはず。余裕があれば茎などの火の通りづらい部分を先に入れるといいです。フライパンを振りつつ強火でガーッと炒めます。炒めすぎると水分が出過ぎてシャキシャキ感がなくなってしまうので、完全に火が通るまでもう少しというところで皿に出してしまうのが実は丁度いいです。

特殊ケース：ほうれん草

ほうれん草は野菜の中でもちょっと特殊で、アクがあるのでただ炒めるだけだと渋い炒め物になってしまいます。炒めてる途中にお湯を入れることでアクを湯に逃がしましょう。油とフライパンを加熱するところまでは他の野菜と同じですが、ほうれん草を入れてざっと混ぜたらお湯を少量 (1 人前で 100ml くらい) 入れてすぐに蓋をして、強火のまま 10-20 秒くらい待ちます。その後ふたを外して全体を炒めて、完全に火が通るまでもう少しのところでザルに出してお湯を捨ててください。 こうすることで下茹で不要の簡単おいしいほうれん草炒めの完成です。

4.2. 香味野菜

例：ねぎ、生姜、にんにく

香味野菜は料理に風味を与えるための野菜です。 大体どんな食材とも合うので重宝します。特ににんにくは保存がききますし、大量にストックしておいて毎回炒め物に使うといいです。

ねぎ

ねぎは適当にスライスして野菜の基本の炒め方をしておけば大丈夫です。カンタン！

生姜、にんにく

生姜やにんにくはメインの食材 (大体野菜になるんじゃないでしょうか) を炒める際に、スライスして油と一緒にフライパンに入れておくとよいです。チューブは切る手間が省けて便利ですが油を加熱する際に飛びちっちゃうのが難点です。まあでも便利なので難点をわかった上で使う分にはいいのではないかと思います。

4.3. 薄切り肉 or 鶏肉

ここでは豚肉か牛肉の薄切り肉、あるいは鶏肉を一口サイズに切ったものを想定しています。なぜなら寄稿者がそういう肉しか調理しないからさ！

カリカリに焼く

一番手っ取り早いのは適当に切った肉をテフロンのフライパン（中華鍋だと焦げつかないようにするの大変なんよね .....）で中〜強火で表面がカリカリになるまで焼くことです。これが

うまいんですよ。鶏もも肉の場合は肉を切る前に皮目を下にして重しを置いて焼くと皮目がパリパリになっていいですよ。

漿 (チャン) でプルプルの肉にする

漿はおすすめのイディオムです。少し手間ですがこれを行うことで肉がプルプルになって炒め物の出来が一段とよくなります。分量は大体こんな感じ

- 切った肉 : 100g
- 溶き卵 : 40g（余った分は炒めちゃいましょう）
- 片栗粉 : 小さじ 1-2 杯
- (お好みで) 塩 : 小さじ 1/4 or 醤油 : 小さじ 1

まず切った肉に卵を吸わせます。溶き卵を少し入れてよくもみ込んで、を数回繰り返して肉に卵をしっかり吸わせましょう。このタイミングでお好みで下味の塩か醤油も入れてしまいます。それができたら片栗粉を入れてこれもよく揉み込んで、最後にテフロンフライパンで炒めて火を通しましょう。

4.4. 挽肉

挽肉は肉味噌にするのがおすすめです。肉味噌はそれ自体に味がしっかりついているので、炒め物に入れる際は他に味つけは不要です。大量に作ってストックすると便利。

肉味噌

ここでは甜麺醤を使ったレシピを乗せておきます。甜麺醤がない？保存効くから買っちゃいましょう。肉味噌でなくても炒め物の味付けに使えますよ。それか味噌を使ったレシピもあるので適当に調べてください（雑）。調味料が変わるだけで基本的には同じです。

- 油 : こさじ 1
- 挽肉 (なんでも): 100g
- 醤油 : 小さじ 1
- 甜麺醤 : 大さじ 1

まずフライパンに油をしいて強火で加熱します。温まってから入れた方が焦げつきにくいで

す。挽肉を入れたら、へらでほぐしながら強火で炒めます。大体火が通ったら醤油と甜麺醤を入れて、よく混ぜながら加熱し続けます。肉から出る液体が透明になったら、それは全部油なので十分に火が通ってます。炒め物に使う分は皿に、保存する分はタッパーに入れましょう。

4.5. 豆腐

豆腐は結構クセ物で、水分をたっぷり含んでいるのでそのまま炒め物に入れようとすると全体がびちょびちょになってしまいます。なので炒める前にキッチンペーパーで包んで重しを置いて半日ほど置いて水を抜くようにしましょう（いや、大変なのはわかるけど実際これで大分出来が変わってくるんや、信じてくれ……）。うまく水が抜ければ手で持っても簡単には崩れなくなっているはずです。手でちぎるなり包丁で切るなりして小さく分けて、油をしいたフライパンでしっかり焦げ目をつけましょう。木綿の方が楽ですが絹でもなんとかなります。

4.6. 卵

先にも言ったように卵は別々に炒めることで美味しく炒めることができます。野菜と同じで強火で炒めるとよいです。まず油をひいたフライパンを強火にかけて充分に加熱しましょう。そこに卵を入れると端から泡立つように焼けますが、半分くらい固まるまで待ちます。その後箸でぐるっとかきまぜつつ、完全に火が通るまで炒めてしまいましょう。[*27]

4.7. 木くらげ

木くらげ (乾燥のやつ) は保存が効いて調理も簡単、食感もよいので常備しといて損はありません。適当な量をお湯で戻しておいて、最後の混ぜる段階で加えれば大丈夫です。

5. まとめとおまけ

以上が炒め物のイディオムの一覧です。4 節にあるのが全てというわけではありませんし、もっと良い調理方法があるかもしれません。試行錯誤で料理の幅を広げていくのも料理の楽しみの一つですので、是非色々試していって、なんだったらインターネットや同人誌で共有してください。

それと最後に読者の皆様が炒め物以外の料理を作りたい時のために、寄稿者が料理を作る際

27 半熟が好きという人もいると思うんですが、炒め物で半熟の卵でうまくやる方法を寄稿者は発見していません。知っていたら教えてください。

にお世話になっているレシピサイトや料理本を紹介させてください。多分ここが本記事で一番大事なところだぞ。

5.1. 白ごはん .com https://www.sirogohan.com

和食を豊富に取り扱っているサイト。レシピ記事が「how」だけではなく「why」まで説明してくれているのでめちゃくちゃ勉強になるし、なによりおいしい。何故こんなにちゃんとしてるサイトが無料なのかわからない。ちなみに関連サイトとして「白ごはん .com ストア」があって料理グッズを買うことができます（https://shop.sirogohan.com）。読者の皆さんにもお礼のつもりでお買い物してくれると嬉しい。おすすめのレシピは「豚の生姜焼き」と「ゴーヤチャンプルー」。

5.2. 「レシピを見ないで作れるようになりましょう」有本葉子

タイトルを見ればわかる通り本記事はこの本の影響をモロに受けて書かれています。食材ごとの料理法を色々紹介して、レシピを見ないでも手元の食材で料理を作れるようにしようという本です。ちゃんと自炊したい人にはこれが一番いいんじゃないでしょうか。おすすめのレシピは「シンプル野菜炒め」[*28]。

5.3. 「四川料理のスゴい人 自宅でつくる本格中華レシピ」人長良次

同名のメシ通の人気連載を書籍化したもの。もとの連載は「四川料理のスゴい人」で検索すると出てきます。本当に家の設備でちゃんとおいしい四川料理が作れます。たまらん。「肉味噌」と「漿 (チャン)」のイディオムはここから拝借しました。おすすめのレシピは「水煮牛肉」。

5.4. 「いちばんおいしい家カレーをつくる」水野仁輔

この水野さんという方はカレーが好き過ぎてカレーの本を多数出版している異常者（褒め言葉）らしいのですが、この方のレシピに従って作るカレーが実際めちゃくちゃうまいです。結構手間はかかりますが実際それだけの価値はあるので休日にちょくちょく作っています。この本では「欧風カレー」と「インドカレー」の二つのレシピを紹介した後に、それらのいいとこどりをしたレシピ修正不要のカレー「ファイナルカレー」を提示するという構成をとっていま

28 「野菜」のイディオムとかなり近いレシピです。でもこの部分がこの本から来ているかというとそうでもなくて、ああいう野菜の炒め物の作り方は結構色々な場所で紹介されています。寄稿者が最初に見たのは土井善晴「ふだんの料理がおいしくなる理由」でした。

す ..... いるんですが、 個人的には「欧風カレー」 が好きです。 というわけでおすすめレシピは「欧風カレー」。炒め物には関係ないんですが、うまいカレー情報を共有したかった。

自作言語 Sesterl でオンライン対局ゲームを実装した話

gfn

1. はじめに

Sesterl（セスタール）[23] という自作言語で**天九 Online** [24] というオンラインゲームを実装した話を報告したいと思います．以下のような前提知識を想定します：

- 静的型つきの函数型プログラミングに関する基礎的な理解.
  - 例えば Haskell や OCaml でプログラムを書いたことがある，単純型の型つけ規則を眺めたことがあるなど.
- Web アプリケーションの仕組みに対する基礎的な理解.
  - クライアント - サーバモデルという概念や HTTP の規格の概要を把握しているなど.

Sesterl は 2020 年 4 月頃から筆者が開発している計算機言語で，簡単に言えば Erlang という既存の計算機言語に静的な型システムを搭載することを目的としているものです．要するに，JavaScript に対する TypeScript, PureScript, Elm, ReScript といった立ち位置の言語を Erlang に関して実装しようという試みであり，AltJS になぞらえて言えば AltErlang という具合の言語です．Erlang は**アクターモデル** [8] に基づいた意味論をもち，並行処理や分散処理に関する機能がビルトインで提供されているなど並行並列に特に強みをもつ言語ですが，静的に型がつかず[*29]，実装の正しさを保証したりリファクタリングを施したりするのにかなり苦労する言語です．これを解決することを企図して開発しているのが Sesterl というわけです．

29 一応 success typing [14] という型システムおよびその型検査器の実装である Dialyzer がありますが，後づけで導入されたものであって健全性 (soundness) を満たすことを志向した型システムではなく，型がつくことによって保証できるプログラムの性質はかなり弱い主張に留まっており，型検査に通っても実行時に形式の不整合でエラーが起きることはごく普通にあります.

後々もう少し詳細に触れますが, Sesterl の型システムの概要を言うと以下のようなものが備わっています：

- **Damas–Milner 多相**（いわゆる let 多相）およびその **Hindley–Milner 型推論**
- **列多相**によるレコード型 [7]
- 函数のラベルつき必須引数，および列多相を利用したラベルつきオプション引数
- F-ing Modules [22] に基づく，ファンクタなどを扱えるモジュールシステム
  - このモジュールシステムは特に（`gen_server` や `supervisor` といった）Erlang の **OTP ライブラリ**と呼ばれるライブラリ群を型安全にラップするのに利用しています.
- 純粋な計算と並行処理とを区別するためのモナド
- Erlang で実装した函数に型註釈をつけて Sesterl 側で使えるようにする FFI

1 年ほど開発を続け Sesterl も或る程度成熟してきたのですが，「そうはいっても本当に実用に堪えるものになっているだろうか」という疑問が湧き，また丁度数年前からいつか実装したいと思っていたゲームがあったので, 2021 年 5 月末頃から Sesterl を用いて実際に稼働するオンラインゲームのサーバサイドを実装してみることにしました．そうして（不足を感じたら適宜 Sesterl の方も拡張しつつ）ひとまず動作するようになったのが天九 Online というゲームです． 天九 Online は**天九**または**打天九**という中華圏で古くから遊ばれている既存の卓上ゲームを実装したもので， これは**天九牌**という 32 枚 1 セットの牌を使ったトリックテイキングゲーム[*30]です． 天九のルールについても簡単に紹介しつつ， 実際に卓上ゲームが対戦できる小さいサーバをどのように設計・実装したか紹介します.

30 概して次のような性質を満たすルールのゲームを**トリックテイキングゲーム**と言います：

- 1 ゲームが複数の**トリック**という小さい勝負からなる.
- 最初はプレイヤー全員に同じ枚数の手札が渡されている.
- 各トリックでは，最初の手番の人から順に各プレイヤーが 1 回ずつ手札から場に札を出し，全員分出揃ったらその出揃ったカードに基づいてそのトリックの勝敗を決める．勝った人が次のトリックの最初の手番となる.
- トリックで場に出たカードのうちどれが勝つかの判定基準は， 最初の手番のプレイヤーが出した札である**台札**によって変わる.
- 最後のトリックが終わったら，それまでの各トリックでの勝敗などをもとに何らかの条件でゲーム全体の勝敗や得点の移動を決める.

いわゆるトランプを使って遊ぶトリックテイキングゲームの例としては**ハーツ**や**ナポレオン**などが有名です.

趣味で言語処理系を実装する方は世の中に数多くいらっしゃると思いますが，大方の自作言語は制作者自身の言語処理系への理解のために創られたものであったりして，実用可能な水準まで作り込む経験談はあまり見かけないように思います．実用を目的としてドッグフーディングで自作言語を作り込むのはチマチマとした改善を継続する地味ながら執念を要する開発ですが，実際に自作言語のおかげで（そうでない場合よりも）高い生産性を発揮できていることが実感できたときの欣快はひとしおです．そんなわけで，ちょっと自己満足的な側面も含んでしまいますが，自作言語を実用してゲームを実装しつつ言語もドッグフーディングで改善していった過程を簡単に共有できればと思い，記事にしました．

2. Erlang の簡単な解説

Sesterl は Erlang という既存の言語をラップするために開発されました．では元々の Erlang とはどんな言語なのでしょうか？ おそらく「Erlang という名前くらいは聞いたことがあるが使ったことは全くない」という読者も多いかと思うので，簡単に Erlang の構文や意味論について説明します．とはいえ Erlang に入門するための文書は公式ドキュメントはもとより非公式のものもいくらでもあるはずなのでここで懇切丁寧に解説するつもりはなく，全く読み書きしたことのない人に最低限どんな言語かの雰囲気を掴んでもらうためにかいつまんで記載します．

2.1. 基本的な構文と意味論

モジュールと函数の定義

前述の通り，Erlang の意味論はいわゆる函数型言語のそれを基調としており，例えば階乗函数 `fact` や整数のリストを受け取りその総和を返す函数 `sum` は以下のように書けます：

```
-module(calc).
-export([fact/1, sum/1]).

fact(N) ->
    case N =< 1 of
        true  -> 1;
        false -> N * fact(N - 1)
    end.
```

```
sum(Ns) ->
    lists:foldl(fun(N, Acc) -> N + Acc end, 0, Ns).
```

変数の名前は大文字始まり，トップレベルで定義される函数の名前は小文字始まり[*31]，函数定義の終わりにはピリオドを打つ，case 式によるパターンマッチングは枝の区切りにセミコロンを使う，他のモジュールの函数を呼び出すには lists:foldl などとモジュール名と函数名をコロンで繋いで書くなどかなりアクの強い Prolog 風の具象構文をもっていますが，Lisp 諸方言，Haskell，OCaml あたりに親しみのある方なら函数定義についてはわりと構文から意図するところが推測できるのではないかと思います．

Erlang ではモジュールを入れ子にできたりはせず，全てのモジュールが平坦に 1 つの名前空間に並び，1 つのソースファイルが 1 つのモジュールの定義を担います．上記の例の 1 行目はこのソースファイルが記述するモジュールの名前を，2 行目はソースファイル内で定義されている函数のうちどれを公開するか（＝モジュール外から呼び出せるようにするか）を記載しています．fact/1 が階乗函数の名前で，/1 はアリティ（＝何個の引数をとるか）を表します．アリティが名前の一部なのは，函数が（Haskell や OCaml などと違い）Curry 化されておらずアリティをもち，かつ同一の名前で異なるアリティの函数は全く別の（たまたま名前が同じ）函数として扱うことを許す言語設計になっているためです．特に，アリティの異なる補助函数は同名にする慣習があり，上記の fact/1 を末尾呼び出し再帰になるように書き換えるときには以下のように補助函数を fact/2 としたりします：

```
fact(N) ->
    fact(1, N).

fact(Acc, N) ->
    case N =< 1 of
        true  -> Acc;
        false -> fact(Acc * N, N - 1)
    end.
```

なお，パターンマッチングやそれによる分岐は函数の引数でも行なえて，また when 節により

31 トップレベルの函数の名前は正確には後述のアトムの形式と一致し，単引用符で括って一般の文字列が名前として使えます．

マッチする条件を加えることができる[*32]ので, `fact/1` はさらに以下のような小慣れた実装に書き換えることができます：

```
fact(N) -> fact(1, N).

fact(Acc, N) when N =< 1 -> Acc;
fact(Acc, N)             -> fact(Acc * N, N - 1).
```

こうして定義した `fact/1` は，モジュール外から使うときは以下のように呼び出します：

```
-module(some_external_module).
-export([main/0]).

main() -> calc:fact(6).
```

局所束縛

局所的な変数の束縛は少し独特な具象構文で，以下のように書きます：

```
distance(X1, Y1, X2, Y2) ->
    XDiff = X1 - X2,
    YDiff = Y1 - Y2,
    math:sqrt(XDiff * XDiff + YDiff * YDiff).
```

変数だけでなく一般のパターンも左辺に書くことができます（$\{p_1, p_2, \ldots, p_n\}$ は組にマッチするパターンです）：

```
distance(Pos1, Pos2) ->
    {X1, Y1} = Pos1,
    {X2, Y2} = Pos2,
    XDiff = X1 - X2,
```

32 ただし, `when` 節の中では限られた組み込み函数しか使えないように制限されています．これは `when` 節の評価中に或る種の “副作用” が発生しないことを保証するためのようです.

```
    YDiff = Y1 - Y2,
    math:sqrt(XDiff * XDiff + YDiff * YDiff).
```

引数を直接パターンにしてもかまいません：

```
distance({X1, Y1}, {X2, Y2}) ->
    XDiff = X1 - X2,
    YDiff = Y1 - Y2,
    math:sqrt(XDiff * XDiff + YDiff * YDiff).
```

函数も束縛でき，呼び出しはグローバルに定義された函数と同様に書けます：

```
sum(Ns) ->
    F = fun(N, Acc) -> N + Acc end,
    lists:foldl(F, 0, Ns).
```

ちなみに，グローバルに定義された函数も単独で函数抽象として使えます：

```
G = fun lists:foldl/3,
```

文字列とバイナリ

Erlang に於ける文字列の扱いは少し特殊で， まず通常の二重引用符で括られた文字列は（Haskell の `String` が `[Char]` に等しいのと似て） 単に “コードポイントのリスト” です． 例えば `"hello"` と書いた場合， これは `[104，101，108，108，111]` と全く等価です． 一方で `<<"hello">>` とさらに二重三角括弧で括って定数を書いた場合，これは “エンコード済みのバイト列としての文字列” を表します．こちらが多くの計算機言語で「文字列」と言った時に指すものにより近い値で，このような形式を Erlang では**バイナリ** (binary) と呼んでいます．これはこの形式がテキストに限らず一般のバイト列を格納する形式であるためで，「バイナリファイル」と言う時の「バイナリ」と同じ気持ちでの命名だろうと思います．

マップ（辞書，連想配列）

Erlang には**マップ** (map) という一般に辞書とか連想配列とも呼ばれる表引き用の機能があり，以下のように `#{`k_1 `=>` v_1`, …,`k_n `=>` v_n`}` という形で記述します：

```
Map = #{<<"太郎">> => 110, <<"敏子">> => 95},
```

取り出しや更新は `maps` モジュールの函数によって行なったり，パターンマッチングによって行なったりします（詳細は省略）．キーの等価性判定は組み込みの等価演算 `=:=` によっており，OCaml でのファンクタ `Map.Make` の引数に実装を与えたりして弄るようなことはできません．

アトム

Erlang には**アトム** (atom) という Ruby に於けるシンボルとよく似た機能があります．すなわち，“一定の決まった文字列で表される定数” です． これは小文字始まりの `foo` とか `bar` といった文字列のトークンで，特にどこかで定義せずともソースコード中でいきなり使えます．OCaml の多相バリアントを引数が取れないように制限したものと捉えてもよいかもしれません．また，小文字始まりでなくても，`'Capitalized'` とか `'@foo'` のように単引用符で括ることでより一般の文字列をアトムにできます．

アトムの典型的な用例は，タプルと組み合わせて代数的データ型のように使うものです．例えばキーでマップを表引きする函数 `maps:find/2` は，対応する値 v が見つかった場合は `{ok,`v`}` を，見つからなかった場合は `error` をそれぞれ戻り値として返します．すなわち，OCaml でいえば前者が `Some` v，後者が `None` に相当する表現なわけです．

Erlang では色々なものが実はアトムとして定式化されており，例えば以下はいずれもアトムです：

- 真偽値，すなわち `true` と `false`．
  - これらは “真偽値専用の値” ではなく，“そういう文字列のアトム” にすぎない．
- モジュール名．
- （アリティを除いた）函数名．

したがって，（実装がスパゲティ化するおそれと隣合わせなので乱用には注意すべきですが）実は函数呼び出しも以下のように書けたりします：

```
ModName = math,
FunName = sqrt,
ModName:FunName(2).
```

パターンマッチング

既にちらっと登場しましたが，パターンマッチングは case 式によって行ないます：

```
tree_size(Tree) ->
    case Tree of
        empty                -> 0;
        {node, Tree1, Tree2} -> 1 + tree_size(Tree1) + tree_size(Tree2)
    end.
```

パターンは Haskell や OCaml に慣れている方にとっては大方想像通りで，_ がワイルドカードなのも同じです．

ただし，1 つだけかなり意外な落とし穴があります．既に束縛されている変数をパターン中で使った場合は，その変数が束縛されている値を書いたことと同じ扱いになります．例えば，以下の定義に基づくと `f()` は `false` に評価されます：

```
f() ->
    X = 42,
    case 57 of
        X -> true;
        _ -> false
    end.
```

実は通常の局所束縛も左辺がパターンなので同じことが言えて，次の定義に基づくと `g()` の評価は 3 行目で「57 が 42 というパターンにマッチしない」という失敗が起き例外が送出されます：

```
g() ->
    X = 42,
    X = 57,
    X.
```

ここからの帰結として，同じスコープでは同名の変数を複数回束縛することができません．それゆえに Erlang コードでは以下のようなちょっとマヌケな連番の変数名をよく見かけることがあります[*33]：

```
update_state(SomeParameter, State0) ->
    State1 = run_first_update_process(State0),
    State2 = run_second_update_process(State1),
    …
    State5.
```

spec 註釈

Erlang にも健全性を満たさない体系ながら success typing という型システムがあり，これを実装した Dialyzer という型検査器があります．この Dialyzer 向けに，或いは人間のコードリーディングの補助のために，大域的に定義された函数には “どんな形式の引数がくることを想定していて，どんな形式の戻り値を返すのか” ということを表した **spec 註釈**というものを書くことができます．spec 註釈は以下のように記述します：

```
-type point() :: {integer(), integer()}.

-spec distance(point(), point()) -> float().
distance({X1, Y1}, {X2, Y2}) ->
    XDiff = X1 - X2,
```

33 もし筆者が Erlang の言語設計を与える立場だったなら，束縛済みの変数を値として使うパターンは単に `X` などと書くのではなく `val X` などと書くような構文を採用して同じ変数名を複数回束縛できるようにしたいと考えただろうなとよく思いますが，まあそんなことを言っても実際の Erlang はそうはなっていないので気をつけて従うしかないですね．そもそもなぜこんな仕組みがあるのかというと，後に出てくる `make_ref/0` などの機構で動的に決まる “トークン” 的な値にマッチさせたいことがよくあって，それに利用したいからだろうと思います．

```
    YDiff = Y1 - Y2,
    math:sqrt(XDiff * XDiff + YDiff * YDiff).
```

アリティが 0 でも型名には () がつきます．これは以下で触れるような spec 註釈中のアトムと区別するためです．Dialyzer はこの註釈と実際の函数の定義に明らかな不整合がないかを検査します．spec 註釈にはもう少しいろいろなことを表す記法があります：

- `term()` で任意の形式を表す（いわゆる Top 型）．
- `{`T_1`, …, `T_n`}` で直積型を表す．
- `foo` などのアトムで “そのアトム単独のリテラル型” を表す．
- T_1 `|` … `|` T_n で合併型を表す．
  - 例えば `boolean()` は `true | false` と等しい．
  - いわゆる result や either は `{ok,` T_1`} | {error,` T_2`}` で表されることが多い．
- `when` を用いて一部分にローカルに名前を与えて括り出すこともできる：

```
-spec distance(Point, Point) -> float() when Point :: {integer(), integer()}.
```

- `{error, Reason :: term()}` のように，わかりやすさのために余剰な名前を加えることもできる．この例は `{error, term()}` と全く同じ．

他にも色々な記法や組み込みで定義された型がありますが，ここでは上記程度の内容で済ませることにします（以降の記事中でも少ししか使いません）．

2.2. 並行処理の定式化

前述の通り，Erlang は並行処理や分散による並列処理を得意とする計算機言語で，特に並行や分散の処理は組み込み函数や専用の制御構文といった言語機能としてサポートされているのが特徴です．Erlang に於ける並行処理は**アクターモデル** (actor model) [8] という定式化に基づいたもので[*34]，これは以下のような特徴をもちます：

34 ただし，Erlang の言語設計を行なった Joe Armstrong らは特にアクターモデルという概念を把握していたわけではなく，結果的に似た定式化になったというのが史実だそうです．このことを指して逸話的に「良い設計は何度も再発見される」と言う人もいたりします．

- 計算は，**プロセス**と呼ばれる互いにメモリを共有しない単位が，複数個並行して各々に与えられた式を評価することによって行なわれる．
  - この「プロセス」は OS レベルのプロセスとは関係のない概念であることに注意．
- 各プロセスは**プロセス ID** 或いは略して **PID** と呼ばれる固有の ID をもち，或るプロセスが別のプロセスにメッセージを送る時の宛先などとして使われる．処理を走らせているプロセス自身の PID は `self()` という組み込みの函数の適用を評価すると取得できる．
- 新たにプロセスを生成する処理は基本的には `spawn/1` という組み込みの函数によって行なわれる．`spawn(fun() ->` e `end)` が評価されることで新たなプロセスが生成され，そのプロセスが式 e の評価を始める．呼び出した側のプロセスには，新たに生成されたプロセスの PID が即座に戻り値として返る．
- 或るプロセスが別のプロセスにメッセージを送る処理は，その送り先の PID を表す式を e_{pid}，メッセージとして送りたい値を表す式を e_{msg} とすると，e_{pid} ! e_{msg} という式を評価することで実行される．この式全体の戻り値としては即座に e_{msg} の評価結果であるメッセージの値が返り，送信は相手の受信を待つなどせずに完了する．すなわち，送信は非同期的である．
- 各プロセスは，自分宛てに送られてきたメッセージを自身の**メールボックス** (mailbox) という機構に届いた順に蓄える．この処理は“プログラム側からは見えないうちに”行なわれている．
- 受信処理の機能としては，`receive` p_1 `->` e_1`;` … `;` p_n `->` e_n `end` という専用の構文がある．プロセスは評価中にこの式に出会うと自身のメールボックスに溜まった値の列 $v_1,\ldots,v_m$ を見て，まず最初に届いていた v_1 について $p_1,\ldots,p_n$ のいずれかのパターンと（`case` 式のように）マッチするかどうか確かめる．p_i とマッチしたら，その中の変数を束縛して，メールボックスから v_1 を取り除き e_i の評価に移る．どれともマッチしなかったら，v_1 をメールボックスの中に先頭位置で残したまま，次に v_2 が $p_1,\ldots,p_n$ のいずれかとマッチするか同様に確かめ，p_i とマッチしたらやはり v_2 をメールボックスから取り除いて対応する e_i を評価する．その後もメールボックスに溜まった値とパターンを照らし合わせていき，最後の v_m までどれもマッチしなかったらプロセスは待機することにし，以後新しいメッセージが届くごとに同様のことを行なう．
  - ただし，この仕組みだけだと何らかの理由で期待するメッセージが一向に届かないとき

に `receive` 式で永久に待機してしまうおそれがあるので，そういう状況が望ましくない場合は `after` e_{duration} `->` $e_{\text{otherwise}}$ という枝を `receive` 式の分岐のひとつとして設けることができる．e_{duration} はタイムアウトを表すミリ秒単位の整数で，この分岐は `receive` で待機し始めてから e_{duration} ミリ秒経っても他のパターンにマッチするメッセージがひとつも来なかった場合は $e_{\text{otherwise}}$ の評価に移行する，という仕組みを提供する．

- プロセスは，自身の式 e の評価が終わる（か評価途中で捕捉されない例外が送出されるかする）と消える．消えたプロセスに向かって e_{pid} ! e_{msg} で送信した場合は，単に何も起こらない．

複数プロセスが並行に走る簡単な例を見てみます：

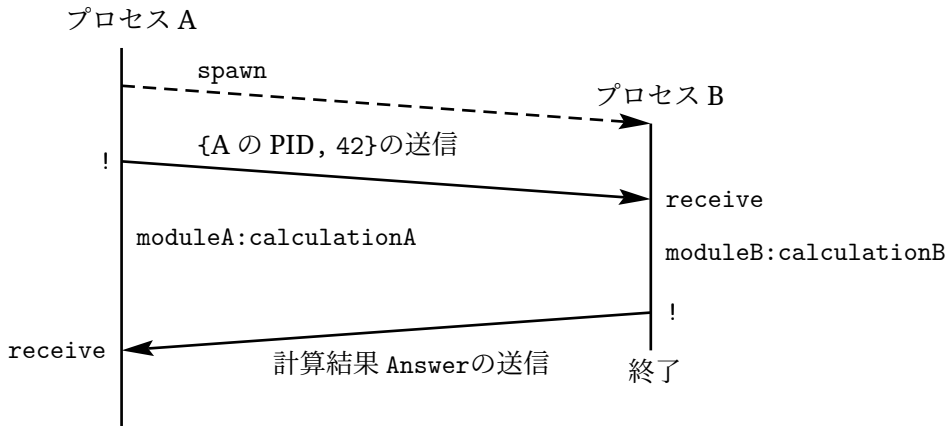
```
-module(concurrency_example).
-export([main/0]).

sub() ->
    receive
        {From, N} ->
            Answer = moduleB:calculationB(N),
            From ! Answer
    end.

main() ->
    PidA = self(),
    PidB = spawn(fun() -> sub() end),
    PidB ! {PidA, 42},
    ResultA = moduleA:calculationA(57),
    receive
        ResultB -> {ResultA, ResultB}
    end.
```

或るプロセス A で `concurrency_example:main()` が呼び出されて処理が開始する想定の実装です． このプログラムでは， まず呼び出したプロセスの PID に `PidA` を束縛し， 続いて新しいプロセス B を生成して `PidB` をその新しいプロセスの PID に束縛します． 起動されたプロセス B では `sub()` が評価され，`receive` 式を評価するところで待機します． プロセス A は返信

用の自身の PID と 42 という整数の組を B に向かって送ります. B はそれを受け取って From を A の PID に, N を 42 に束縛し, moduleB:calculationB(42) を評価します. その間に A は moduleA:calculationA(57) を評価します. B は計算が終わったらその結果を From ! Answer で A に送り返して終了します. A も計算が終わったら自分のメールボックスを見て B からの返信が届いているかを確認し, 届いていたら（或いは待機して届いたら）ResultB にその結果を束縛し, 最終結果の {ResultA, ResultB} を main() の戻り値とします. 要するに, プロセス A が moduleA:calculationA(57) を, プロセス B が moduleB:calculationB(42) をそれぞれ並行して処理するような実装になっているわけです. この並行処理の動作の様子をシーケンス図に描き起こすと以下のような具合です[*35]：



2.3. OTP Design Principles について

並行処理がどのように実現されているかは前節で紹介しましたが, 実際には逐一 spawn, !, receive といった比較的低級な送受信の実装を直接記述する必要はあまりなく, OTP（Open Telecom Platform） と呼ばれるライブラリ群を使って並行処理を実現することが多いです. OTP は並行処理, とりわけ可用性の追求を見越した並行処理に於いて共通の典型的なパターンを抽出したライブラリで, その中でも gen_server モジュールと supervisor モジュールを利用することがほとんどです.

35 メッセージがメールボックスに入るタイミングとメールボックスに入っていたメッセージが receive 式によって取り出されるタイミングは一般には異なるので図はやや不正確ですが, おおよその直観を汲み取っていただけると幸いです.

gen_server モジュール

並行処理のプログラムを書くと, “状態を保持して長期間存在し，適宜外部とやり取りして状態を更新したり，外部から状態の一部を取得されたりするプロセス” というパターンがとても多いことに気づきます．というかこのパターンに収まらない例が珍しいくらいかもしれません．このパターンにあてはまるプロセスとして，状態として唯一つ整数を保持し，外部から整数が取得されたり更新されたりするもの（以下では**セルプロセス**と呼ぶことにします）を扱うことを考え，それを `cell` という以下のような API のモジュールとして実装することを考えます：

- `-spec start(integer()) -> pid().`
  - 初期状態を与えてセルプロセスを起動し，その PID を返す．
- `-spec get(pid()) -> integer().`
  - セルプロセスが現在保持している整数を取得する．
  - これはセルプロセスからの返信が必要な処理なので返信が来るまで待つ．既にセルプロセスが存在しない場合や, 5 秒経っても返信がない場合は例外送出とする．
- `-spec set(pid(), integer()) -> ok.`
  - セルプロセスが保持する整数を更新する．
  - これは指示を出すだけで返信が要らないので送信側はブロックされず，即座に戻り値の `ok` が返る．
- `-spec stop(pid()) -> ok.`
  - セルプロセスを終了させる．

これを実装したのが以下のようなものです（コード中の `make_ref/0` は初出ですが，後ほど説明します．`throw/1` は詳しくは説明しませんが例外送出です）：

```
-module(cell).
-export([start/1, get/1, set/2, stop/1]).

start(N) ->
    spawn(fun() -> loop(N) end).

get(Pid) ->
    Tag = make_ref(),
    Pid ! {get, self(), Tag},
```

```
    receive
        {got, Tag, N} -> N
    after 5000 ->
        throw({failed_to_get, Pid})
    end.

set(Pid, N) ->
    _ = Pid ! {set, N},
    ok.

stop(Pid) ->
    _ = Pid ! stop,
    ok.

loop(N) ->
    receive
        {get, From, Tag} ->
            From ! {got, Tag, N},
            loop(N);
        {set, M} ->
            loop(M);
        stop ->
            ok
    end.
```

loop/1 がセルプロセスで稼働する処理で，その引数がセルプロセスの保持している状態にあたります． セルプロセスはこの函数の receive で外部からのメッセージの到達を待機し， メッセージを受け取ったら以下のように対処して，終了する場合以外は loop/1 を再帰的に呼び出して再び待機状態に戻ります：

- 受け取ったメッセージが {get, From, Tag} という “リクエスト” だった場合， 現在保持している整数 N を使って {got, Tag, N} という “レスポンス” を送信元に返す．
    - ここで Tag は “リクエスト” と “レスポンス” を紐づけるために使っている “トークン” で，make_ref/0 は呼び出しのたびにこのような “トークン” をフレッシュに生成する組み込みの函数です．

- 受け取ったメッセージが {set, M} だった場合，保持していた整数を捨てて新たに M を保持するようにする．
- 受け取ったメッセージが stop だった場合，直ちに終了する．

get/1, set/2, stop/1 はそれぞれ {get, From, Tag}, {set, M}, stop のメッセージをセルプロセスに送るものとして実装されており， 他のプロセスで呼び出すことで該当のセルプロセスとやり取りすることができます． このうち特に get/1 はセルプロセスからの “レスポンス” を {got, Tag, N} のパターンで受け取るようになっており， この Tag は束縛済みの変数なので変数パターンではなく値としてマッチすることに注意してください．これにより，たまたま他に同じ形式のメッセージが届いていても， 該当の “リクエスト” と対応する “レスポンス” だけをメールボックスから取り出すことができます．

こうした挙動を記述したい場面はとても多く，OTP ライブラリはこのような実装のパターンを一般化して記述するための仕組みとして gen_server を提供しています．gen_server は， 端的に言えば “受け取ったメッセージに対してどう対処するかだけ与えれば上記の loop や start などに相当する機能（を大幅に拡張したもの）を担ってくれる” モジュールです．まず，モジュールを “gen_server に準拠した実装” にするために, “受け取ったメッセージに対してどう対処するか” などに相当する以下の 5 つの函数を定義します：

- -spec init(InitArg) → {ok, State} | ….
  - 生成されたプロセスが起動直後に行なう処理．少し後に紹介する gen_server:start や gen_server:start_link に与えられた引数 InitArg を受け取り， 初期状態 State をつくって {ok, State} を返す．
- -spec handle_call(State, From, CallMsg) -> {reply, Reply, State} | ….
  - 後述する gen_server:call によって送信されてきた， 返信が必要なメッセージを受信したときに行なう処理． 返信内容を Reply， 更新した状態を State として，{reply, Reply, State} を返す．
- -spec handle_cast(State, CastMsg) -> {noreply, State} | {stop, Reason} | ….
  - 後述する gen_server:cast によって非同期的に送信されてきた，返信不要のメッセージを受信したときに行なう処理．更新した状態を State として {noreply, State} を返す．メッセージをうけて停止する場合はやはり {stop, Reason} を返す．
- -spec handle_info(State, Info) -> {noreply, State} | {stop, Reason} | ….

    - handle_cast とほぼ同じだが，gen_server:cast によって送信されたものではない（! によって送信されてきたものなど）その他のメッセージをさばく処理．後述する**モニタ**による {'DOWN', …} の形式のメッセージを受け取ったりするのに使う．
- -spec terminate(State, Reason) -> ok.
    - （突然死などを除き）終了することが確定したとき，終了直前に呼び出される処理．基本的には何もせず ok を返すのでよい．

これらの函数を定義したモジュール X がある下で，gen_server:start(X, e_{InitArg}, []) を呼び出すと， 上記の函数を使用したプロセスを立ち上げることができます[*36]． 大抵の場合はこの gen_server:start/3 の呼び出しも同一モジュール X の 1 函数としてラップしてしまい，InitArg の形式やモジュール名などをモジュール X の外に対して隠蔽します．モジュール内でそのモジュールの名前を指すには直接書かずとも ?MODULE というマクロが使えるので，殆どの場合は以下のような要領の函数がモジュール内に定義されます：

```
start(...) -> gen_server:start(?MODULE, ..., []).
```

なお，-module(X). の直後に -behavior(gen_server). と追記することで，“gen_server に適合させる” のに必要な init/1 や handle_call/3 など上記 5 つの函数が揃っているかをコンパイル時に検査してくれます．こうした “函数を要求する機構” を**ビヘイビア** (behavior) といい，ビヘイビア B が要求する函数を B の**コールバック函数** (callback function) といいます．例えばこの節では gen_server ビヘイビアが init/1 や handle_call/3 といったコールバック函数を要求することを紹介しました．

ビヘイビア B が要求するコールバック函数を全て実装したモジュールを B の**コールバックモジュール**といいます． セルプロセスを gen_server ビヘイビアのコールバックモジュールとして実装すると以下のようなプログラムになります：

```
-module(cell).
-behavior(gen_server).

-spec start(integer()) -> pid().
start(N) ->
```

36 gen_server:start/3 の第 3 引数には起動オプションを渡すのですが，大抵は空リストで問題ありません．

```
    {ok, Pid} = gen_server:start(?MODULE, N, []),
    Pid.

-spec get(pid()) -> integer().
get(Pid) ->
    {got, N} = gen_server:call(Pid, get),
    N.

-spec set(pid(), integer()) -> ok.
set(Pid, M) ->
    gen_server:cast(Pid, {set, M}).

-spec stop(pid()) -> ok.
stop(Pid) ->
    gen_server:cast(Pid, stop).

init(N) -> {ok, N}.

handle_call(N, _, get) -> {reply, {got, N}, N}.

handle_cast(N, {set, M}) -> {noreply, M};
handle_cast(_, stop)     -> {stop, normal}.

handle_info(N, _) -> {noreply, N}.

terminate(_, _) -> ok.
```

返信を要する同期的メッセージの形式 `CallMsg` が `get`，それに対する返信の形式 `Reply` が `{got, integer()}`，非同期的メッセージの形式 `CastMsg` が `{set, integer()} | stop` というわけです．`{stop,…}` に与えている終了理由の値 `normal` は正常終了を表すための値の 1 つです．

もとが簡単な挙動のプロセスだったのであまり便利になった気がしないかもしれませんが，このような “メッセージを受け取って状態を更新する reducer” を与えさえすればプロセス生成や並行処理を全部担ってくれる `gen_server` による共通化には以下のような恩恵があります：

- 個別に実装することによる不具合の発生を防ぎやすく，プロセスが担う機能の複雑化に対して開発が相対的にスケールしやすい．

- gen_server によってつくられるプロセスは通常の spawn によって起動する “生の Erlang の” プロセスに比べ種々の強化機能を備えたものになっており，それにより後の節で述べる**監視ツリー**に準拠して異常終了をしていないかなどを監視することができる．

プロセスの登録名

プロセスを指し示す方法としては，PID だけでなく**登録名** (registered name) を使うこともできます．登録名は特定のプロセスに紐づいている値で，プロセスが終了すると登録名との紐づきが自動で解除されます．複数のプロセスを同一の登録名に紐づけることはできず，基本的に先勝ちです．

gen_server に準拠したプロセスに名前をつける場合は，起動時に gen_server:start/3 の代わりに gen_server:start/4 を使い，増えた第 1 引数に起動されるプロセスの名前となる値を指定することができます．第 1 引数に与えられる形式は以下のいずれかです：

- {local, atom()}
  - 単一の**ノード**（後述しますが分散の単位となるサーバのことです）の中でのみ通用する名前で登録する場合に使う． 標準の API としてはアトムしか指定できない． アトムに限らない一般の値を使いたい場合は via と専用のモジュールを使うなどする．
- {global, term()}
  - クラスタを構成する全ノードで通用する登録名を登録する場合に使う．こちらは一般の値が名前に使える．
- {via, Module :: atom(), ViaName :: term()}
  - 特定のモジュール Module によって登録名を管理する場合に使う．

gen_server:start/4 で起動する際， 既存のプロセスと登録名が衝突した場合は起動自体に失敗するようになっています．

プロセス間の監視という概念

ここまでの説明では殆ど触れてきませんでしたが，Erlang および OTP は慣習的に「異常が起きたらプロセスごと死ぬようにし， それを監視している別のプロセスが異常終了を検知して場合によっては再起動し再処理を促す」 という設計をする際に便利なようにつくられています．これによって可用性の高いシステムの構築に貢献します．

そもそもプロセスの意図しない終了すなわち異常終了には以下のようなものがあります[*37]：

- 特定の状況で再現性がある意図しない挙動：
  - 特に並行性に関連しない，比較的単純な実装ミス．
- 確率的に失敗し，必ずしも再現しない挙動：
  - 並行処理のスケジューリングに依存して稀に顕現する不具合． すなわち， 大抵は意図通り動くが，実はデータ競合を起こす実装になっていて稀に意図していない挙動をする場合[*38]．
  - プログラムに問題はないが，マシンの性能の局所的悪化により I/O が詰まったり分散処理に失敗した場合．

前者は単にテストや Dialyzer の検査によって発見することが想定されており， まあこれは Erlang 以外の大抵の言語で同様であろうと思います．Erlang が独特なのは後者のような確率的に失敗する場合への対処です．大抵の言語だとデータ競合は静的に起きないことを保証しようとしますし，おそらく計算機言語の理論家ならそういう言語設計を自然と志向するのではないかと思いますが, Erlang はそうではありません．Erlang では「典型的には正しく動くが，確率的に稀に失敗する」 という状況は “普通に起きるもの” として織り込み済みという世界観を取ります．そして，こうした可能性を静的に排除するという方針ではなく，「まあそういうこともたまにあるよね，そしてたまにしか起きないなら起きた場合ももう 1, 2 回やり直すうちにいずれうまくいくよね」とおおらかな気持ちで対処するような仕組みを構成しやすい言語設計になっています．なんだかむず痒い気持ちになると思いますが, “不具合は気づかないうちに混入してしまうのが当たり前であると想定しておいて次善の策を取りやすくしている” という点では一定の納得感のある方針です．また，（本稿では詳しくは触れませんが）Erlang では重要な機能として並行処理のほかに並列分散処理があり，**ノード** (node) と呼ばれる単位で複数のサーバがネットワークで繋がり，それらの間で Erlang の値を “じかに投げ合える” **クラスタ**を形成するようになっています．こちらは並行処理と違ってさらに相手ノードが障害で落ちた場合を考慮する必要があったり，ノード間の通信だと先に投げたメッセージが先に届くことを保証できなかったりと不確定要素が大きくなり，扱いがさらに難しくなりますから, “小さい不具合はあ

37 Bohr と Heisenberg からもじって，前者を**ボーアバグ** (Bohrbug)，後者を**ハイゼンバグ** (Heisenbug) と呼んだりします．ハイゼンバグは確率的なものとは限らずより広範に「再現しようとすると消える不具合」全般を指し，例えば「`printf` を挟むとなぜか消える不具合」などを含むこともあるようです．

38 実はプロセス間でのメモリ共有がない Erlang の意味論でもデータ競合は発生します．典型的にはプロセスの開始や終了のタイミングに関してデータ競合が起きたりします．

るものとして織り込み済み” という方針はなお一層の納得感があります[*39]．ただし，「再起動して再処理すればいいよね」といっても，稼働していたプロセスの状態を復元するのは一般には言語や OTP ライブラリ側ではなく，プログラムを書く側の責任範囲なので注意せねばなりません．すなわち，復元に真に必要なデータがある場合は，より信頼性の高いメモリやストレージに永続化しておく必要があります[*40]．ともあれ，いずれにせよこうした考えにより Erlang および OTP では「プロセスがプロセスを監視する」という概念が導入されているのです．

リンクとモニタ

前節でプロセス間には監視する・監視されるという関係があることを説明しましたが，それがどのように言語機能および OTP ライブラリで定式化されているのかを紹介します．まず，言語機能としての監視にまつわる関係には以下の 2 種類があります：

- **リンク** (link)
- **モニタ** (monitor)

リンクは 2 つのプロセス間に双方向に張られる関係で，リンクしている一方のプロセスが終了すると，他方のプロセスもそれにつられて（何らかの処理中であろうと）自動で終了させられるというものです．リンクが 3 つ以上のプロセスで連鎖的に張られてていると全部のプロセスが連鎖的に終了するので，基本的には “何らかの経路でリンクにより繋がっている複数のプロセスは全部ひとかたまり” と考えてよく，その意味で基本的にリンクは “プロセスの集合を同値類で割る機構” です．このリンクという機能は “ひとつでも欠けると機能を果たさなくなるプロセスの集合” の間で張り合い，正常終了か異常終了かにかかわらずどれかが終了したら自動的に全部終了するようにする仕組みです．

リンクは既に起動したプロセスに対して設定や解除をすることもでき，具体的には `link/1` や `unlink/1` といった組み込み函数に相手プロセスの PID を渡すことによって設定 / 解除できますが，大抵の用例およびライブラリの定式化ではプロセスの起動時に起動元のプロセスから “自分とリンクして起動するか否か” を指定し，以後それらのプロセス間のリンクの有無は不変で

---

39 Erlang の開発は Ericsson というスウェーデンの通信機器メーカーによって 1980 年代に行なわれたもので，実際に電話交換機を Erlang で実装するチームからのフィードバックによって迅速に言語設計が拡張されていたそうです．電話交換機を実装するためにできる限り分散してスケールするようにし，かつ通信を途絶えさせないために可用性を重視して不具合にも対処せねばならないことを念頭に置いてこのような言語設計になったという点はバックグラウンドからも理解しやすいかと思います．

40 この目的のために使えるものとして，ETS および DETS という Erlang のランタイムに内蔵されたそれぞれ揮発性・不揮発性のテーブルがあったりします．

す．低級な機能としては，組み込み函数の spawn/1 を spawn_link/1 に置き換えて使うと呼び出したプロセスとリンクした状態で新しいプロセスを起動することができます*41．より高級な API，例えば gen_server に準拠したプロセスなら，起動処理として gen_server:start/3 ではなく gen_server:start_link/3 を使うことでリンクして起動することができます． 以前の節の gen_server の紹介では説明を簡単にする都合で避けましたが，gen_server:start/3 よりもむしろ gen_server:start_link/3 を使うことの方が典型的です．

一方，**モニタ**は単方向的な関係で，或るプロセスが或る別のプロセスの終了を一方的に検知するのに使います． あるプロセス A とプロセス B があるとして，B の PID を表す式を e_{pidB} と表すことにすると，monitor(process, e_{pidB}) という形式で組み込み函数 monitor/2 を呼び出すことで A から B にモニタを張ることができます．この呼び出しの戻り値はモニタを張ることに伴って生成された**モニタ参照** (monitor reference) と呼ばれる或る種のトークンで，次の段落で用途を説明します．

プロセス A からプロセス B にモニタが張られていると，正常終了か異常終了かにかかわらず B が終了した時に A に {'DOWN', v_{monitor}, process, v_{pid}, v_{reason}} という形式の特殊なメッセージが送られてきます．v_{monitor} がモニタ参照で，これによりどのモニタ関係に起因するメッセージなのか判別できます．v_{pid} は終了した B の PID，v_{reason} は終了時の理由です （正常終了なら normal などになっています）．A はこのメッセージの受信を以て B が終了したことを知れるわ

41 spawn_link/1 は一見 spawn/1 と link/1 とを組み合わせて実装できると思うかもしれません．具体的には以下のような具合です：

```
spawn_link(F) ->
    Pid = spawn(F),
    _ = link(Pid),
    Pid.
```

または以下でも良さそうに思えるかもしれません：

```
spawn_link(F) ->
    ParentPid = self(),
    spawn(fun() -> _ = link(ParentPid), F() end).
```

しかし，実はいずれもデータ競合の可能性を孕んでいます．前者は起動元のプロセスで link(Pid) が評価されていないうちに新しいプロセスが起動直後に終了してしまうと起動元のプロセスが一緒に死にませんし，後者は新しいプロセスが link(ParentPid) を呼び出す前に起動元のプロセスが死んでしまうと新しいプロセスが一緒に死にません． このデータ競合を是正するために spawn_link/1 は別個に組み込み函数として用意されているようです．既に述べたように Erlang はわずかな可能性のデータ競合に関しては不整合が起きた時に再処理すればいいという方針ですが，いくらなんでもそのための監視に関わる機能自体がデータ競合を孕んでいては制御しきれないからか，spawn_link/1 が用意されたのではないかと思われます．

けです.

モニタ関係は破棄することもできます. その関係に対応するモニタ参照を v_{monitor} として, `demonitor(`v_{monitor}`)` を呼び出すとそのモニタ関係が解除されます. なお, プロセス 2 つの間に同一方向のモニタを複数個張ることもでき, 複数のモニタ関係がある状況でモニタ先のプロセスが終了した場合は全てのモニタ関係に対応する `{'DOWN',...}` のメッセージが送られてきます.

実際には一定の割合で後述する**監視ツリー**および `supervisor` という機構にラップされた形で使いますが, 基本的にはここで紹介したリンクとモニタという機構がプロセス間の監視の根本を担っています.

`trap_exit`：リンク機能のバックドア

ややこしいことに,「リンクは双方向関係」という点には例外があり*42, 実際には単方向関係のように使うことができます. 各プロセスは「リンクしているプロセスが終了してもそれに伴って自動で終了せずメッセージとして捕捉するモード」を設定することができるのです. このモードはデフォルトでは無効ですが, `process_flag(trap_exit, true)` という組み込み函数の呼び出しによって有効にできます. 大抵はこのモードの有効 / 無効を切り替えたりはせず, 起動直後に設定してそのままということが多いです.

このモードを有効にしていると, リンクしているプロセスが終了したときに自身は終了せず, `{'EXIT',` v_{pid}`,` v_{reason}`}` というメッセージが届きます. v_{pid} はリンク関係にあった終了したプロセスの PID, v_{reason} は終了理由の値です. このメッセージをモニタの場合の `{'DOWN', ...}` と同様に受け取って処理するわけです*43.

監視ツリーと `supervisor` モジュール

さて, プロセス間には監視する側・される側という関係が設けられていることを既に解説し

---

42 例外と言いつつ結構使われます. 後述の `supervisor` の実装にも使われています. 単方向の関係はリンクから取り除いてモニタに集約する意味論に改めた方が自然ではないかという提案の論文もあります [25] が, 今のところそのような破壊的変更には至っていないようです.

43 ちなみに, 以降では触れませんがさらにややこしいことに例外の例外があります： 終了理由を `kill` というアトムにして異常終了したプロセスは, `trap_exit` のモードが有効か否かにかかわらず問答無用で直接リンクしているプロセスを全て終了させます. これによって終了させられたプロセスの終了理由は `killed` になるので, この効果はリンク先のリンク先以降には波及しません. もうここまで来ると人間に制御するのは困難だろうという気がしてきますが, 一応機能としてはこういうものがあります.

てきました．この関係は，“システム全体として大局的に見た時”にはいわば“より正しく動きそうな側のプロセスがより失敗しそうな側のプロセス（の集合）を監視する”という関係でなければ仕組みとして成り立ちにくいはずです．ということは，プロセス間の監視する / される関係は“どちらがより信頼できそうか”の順序で階層構造をなしていると考えることができそうです．実際，おおよそこの階層構造を形作るために使える仕組みが OTP にはあり，それが**監視ツリー** (supervision tree) と呼ばれるものです．

監視ツリーはその名の通り監視する / される関係がなす木構造であり，この監視ツリー上で中間ノードであったり或いは稼働中の状況に応じて子を生成して中間ノードになるプロセスは**スーパーバイザ** (supervisor) と呼ばれます．スーパーバイザは監視ツリー上で子にあたるプロセスたちを監視し，それらの終了を検知したら，あらかじめ指定された方法にしたがって再起動したり，或いは何もしなかったりします．スーパーバイザは `supervisor` モジュールによって実装することができます．

`supervisor` も `gen_server` と同様にビヘイビアとして定式化されており，`init/1` というコールバック函数をコールバックモジュールに要請します．この `init/1` は `gen_server` が要請するそれとよく似ていて，`supervisor:start_link` が呼び出されてスーパーバイザとなるプロセスが起動された直後に呼び出されます．`init/1` が返すべき戻り値には，（正確な形式はやや込み入っているので公式ドキュメント [5] に任せて省きますが）おおよそ以下のようなことを指定できます：

- スーパーバイザがどんな子プロセスをどう起動するか．
  - 固定のいくつかを最初に起動して常に監視するのか，或いは最初は 1 つも子プロセスをもたず動的に子プロセスを増やすのか．
  - 動的に増やす場合，子プロセスをつくってほしいと外部から要請があったときにどの函数を使って起動するか．
- 子プロセスのどれかが死んだときにどう再起動するか．
  - 子プロセスを個々に監視して 1 つが死んでも他には影響なくその 1 つだけ再起動するのか，1 つが死んだら全部の子プロセスを一旦終了させて再起動するのか，或いはどれも一旦死んだらもう再起動しないのか，など．

動的に子プロセスを増やせるスーパーバイザの場合は，そのスーパーバイザの PID または登録名を表す式を e_{supPid} として，外部のプロセスから `supervisor:start_child(`e_{supPid}`, ...)` などと呼び出すことで生成できます．動的に子プロセスを増やしたい場合というのは，例えば Web

サーバを実装する場合に, HTTP リクエストがクライアントから来るごとに 1 つそのリクエストを捌く用にレスポンスを返すまで稼働するプロセスを立ち上げる目的で単一のスーパーバイザを用意する場合などです.

以上のスーパーバイザの仕組みによってプロセスは監視ツリーをなし，意図せず終了したりしても必要なら再起動するというソフトウェアアーキテクチャになっているというわけです. 実際には簡単な木構造に監視関係をあてはめることだけではシステムの振舞いとして足りないこともよくあり，監視ツリーとは別個に “木構造上の水平方向に” リンクやモニタを張ることもよくあります（特にモニタが顕著です）.

3. Sesterl の開発動機

3.1. 型システムの必要性

ここまで前提知識として Erlang の解説をしてきましたが， ここからようやく Sesterl の話です. Sesterl の開発の動機となった Erlang での不満な点を挙げるなら， まずなんといっても Erlang プログラムには事実上静的に型がつかないことです. Erlang には Dialyzer という型検査器がありますが， この型検査器の元になっている success typing [14] という体系は謂わば後づけ的に Erlang に型システムを導入する都合で健全性を志向していない型システムであるため， 型検査に通っても実行時にはごく普通に （健全な型システムがあれば弾けそうな） 形式のミスマッチによるエラーが出ます.

実行時に形式の不整合でエラーが出る可能性が機械的に排除できないことそれ自体も一応厄介な点ではありますが，それはまあテストの水準で比較的潰せるのでよいとして，型がつかないことで真に厄介なのは， それが修正やリファクタリングを極めて困難にする点にあります. 健全性を満たす型システムの備わった言語なら，例えば或る函数定義の引数や戻り値の形式を変えたらそれに伴ってその函数を使っている箇所全てで型検査時にエラーが出るようになり，そのエラーに従えば網羅的に修正できるはずです. しかしながら，健全性を満たす型システムのない状況だとそのような修正が機械的には補助されず，基本的には人力で修正すべき箇所を探し当てるか，或いは旧来の形式も後方互換性のためにサポートし続ける羽目になってしまいがちです. 一応その函数に対応するテストをしっかり用意できていればテストが落ちることによって確認できますが， テストも人間が用意するものなので網羅されている保証はありませんし， 型のつかない言語で走らせたテストの失敗は型検査のエラーに比べると何が原因で失

敗しているのかが些細な形式のミスに対してあまり解りやすい結果になるとは言えない傾向が顕著です．些細なミスとは例えばどこかで 1 要素のリスト `[expr]` にすべきところにその要素 `expr` だけを書いてしまったとか，或いは `{{x, y}, z}` とすべきところを `{x, {y, z}}` と書いてしまった場合などであり，こうしたミスの原因を特定して修正するのにかなりの労力を要してしまう状況によく見舞われます．

このような不満をバッサリ解決しつつ，並行処理が得意という Erlang の利点を活かしてプログラムを書きたいというのが，Erlang に（健全性を満たす）型システムを用意する最大の動機です．

3.2. Sesterl の目的と要件

さて，前節では単に型をつけたい旨を説明しましたが，自分が何を欲しがっているのかという目的の詳細を詰めると以下のようになりました[*44]：

- 既に Erlang で書かれているプログラムを Sesterl から利用しやすくするために，かつ既に Erlang で書かれているプログラムを段階的に Sesterl に移植していけるように，わかりやすい FFI を実現したい．
- Erlang らしいソフトウェアアーキテクチャをそのまま書けるようにしたい．特に OTP ライブラリはそれ自体には手を加えずに Sesterl 側から実装をできるだけ自然な形で利用できるようにしたい．
- 函数の引数がやたらと多かったり或いは引数の一部がオプショナルなものになっていたりしても，どの引数がどうやって与えられるのかをなるべく捉えやすいようにしたい．
- `spawn`, `!`, `receive` などの低級な通信もできるだけ Erlang の意味論に近い形で実現したい．
- 例外処理は実行パスを極端に複雑化させ収拾がつかなくなりやすいので，例外が発生したらキャッチされずプロセスが異常終了するということをひとまず前提にしたい．
- ビヘイビアなどによって依存関係が逆転することもあるので，Erlang 側からも Sesterl で実装された函数を簡単に呼べるようにしたい．
- むやみに通信を起こすプログラムができあがってしまうスパゲティ化を防ぐために，純粋

44 実際には開発当初から明瞭な言語化をしていたわけではなく，おおよそこうしたいと思って設計・実装しているうちに少しずつ煮詰まってきた目的と要件ではあります．今後の進展によってさらに変わることも勿論あるだろうと思います．

な計算と並行処理が発生する計算とを型の水準で区別できるようにしたい．また，可能なら“送受信の振舞い”も型の水準で表現できるようにしたい．

こうした目的設定をうけて，要件を以下のようにすることとしました：

- 基本的にはML風またはErlang風の言語設計とする．
- FFIの簡便さのため，コンパイルのターゲットとしては直接BEAM（＝Erlang VMのバイトコード）を出力するのではなくErlangソースコードを出力するものとする．
- やはりFFIで函数を定義したり呼び出したりするコードを単純に書けるようにするため，ML系言語やHaskellのような部分適用は導入せず，函数はアリティをもつことにする．
- 函数の引数の形式としては通常の引数のほかにラベルつき必須引数とラベルつきオプション引数を用意する．
- OTPライブラリを型の水準で自然に定式化するために，ファンクタをもつモジュールシステムを導入する．
- 通信が発生する計算はモナドでくるむようにし，各並行処理ないしプロセスは“自分がどんな型のメッセージを受け取れるか”の情報をトラックする．これを用いて`spawn`，`!`，`receive`相当の言語機能に型をつける．可能ならsession type [9]やmultiparty session type [10]を備えつける．

3.3. 他のAltErlang言語

やはり人間皆似たことを考えるのか，同様の動機でErlangに型をつけようとして創られた言語はSesterl以外にもいくつかあります：

- Alpaca [19]
  - OCamlやElmに近い構文と意味論をもち，各プロセスがどんな型のメッセージを受け取れるかの情報を静的に検査することができます．
- Gleam [20]
  - Rust風の構文と意味論をもつAltErlangです．函数はアリティをもち，ラベルつき引数などの機能ももっています．メッセージにはコア言語では型をつけない方針で，ライブ

ラリの水準で型つけを実現する方針のようです．2021 年 11 月現在非常に精力的に開発されており，つい最近 JavaScript バックエンドも実装されたようです．

- Caramel [15]
  - OCaml 互換の構文で開発されているものです．PID はメッセージの型をパラメータにとりますが，モナドによる純粋・並行の区別はないようです．
- Hamler [16]
  - Haskell 風の構文と意味論をもつ AltErlang です．型クラスを備えていたりします．並行処理に関するモナドがありますが，メッセージの型をパラメータにとるわけではないようです．また，OTP ライブラリのビヘイビアは型クラスで定式化されているようです．

Sesterl の開発を構想した当時，Alpaca と（当時の）Gleam がどんな言語であるか試したのですが，当時の自分は特にコア言語の水準でメッセージに型をつけて将来的に session type を入れることを重視していたので，これらとは別に新たに開発を開始することとしました．

1 年半ほど開発して，メッセージに対する型つけも簡潔かつ或る程度便利なものになりましたが，session type は結局それほど強い必要性を感じることがなかったため今に至るまで入っておらず[*45]，結果的には（後で触れますが）OTP ライブラリを型つけするための F-ing Modules に基づくモジュールシステムがとりわけ顕著な独自性となりました．

4. Sesterl の言語設計

4.1. 基本的な構文

まず基本的な構文は以下のような例でわかると思います：

```
module Calc :> sig
  val fact : fun(int) -> int
  val sum : fun(list<int>) -> int
  val swap<$a, $b> : fun({$a, $b}) -> {$b, $a}
```

45 もともと “Sesterl” という名前は “a session-typed Erlang” に由来してつけたものだったので，現状としてはちょっと歪な命名になってしまった感があります．

```
  val have_same_length<$a, $b> : fun(list<$a>, list<$b>) -> bool
end = struct
  open Stdlib

  val fact(n) =
    let rec aux(acc, n) =
      if n <= 0 then acc else aux(n * acc, n - 1)
    in
    aux(1, n)

  val sum(ns) = List.foldl(fun(acc, n) -> acc + n end, 0, ns)

  val swap({x, y}) = {y, x}

  val rec have_same_length<$a, $b>(xs : list<$a>, ys : list<$b>) : bool =
    case {xs, ys} of
    | {[], []}                 -> true
    | {_ :: xtail, _ :: ytail} -> have_same_length(xtail, ytail)
    | _                        -> false
    end
end
```

```
require Calc

module SomeModule = struct
  val main() = Calc.sum([3, 1, 4, 1, 5, 9, 2])
end
```

構文はおおよそ ML 系のものを踏襲していますが，以下のような特徴があります：

- ソースコードのファイルは 1 つのモジュールの束縛 `module` X `=` `struct` … `end` であり， シグネチャが必要ならモジュール名の直後に `:>` `sig` … `end` と追記することができる．
- ファイルの先頭に `require` X と書くことで同一パッケージ中の別ファイルであるモジュール X を読み込める． 依存する別パッケージが提供するモジュールは，設定ファイルに記載

しておくとどこでも使えるようになる．例えば上記の例では Stdlib が別のパッケージに由来するモジュールで，open Stdlib で中身が取り出されているので少しわかりにくいが，Stdlib.List.foldl が使われている．

- ストラクチャのメンバとなる（つまり struct … end 直下の）大域的な束縛は val で始め，函数内の局所的な束縛は let で行なう．let は函数のほかに let n = 42 in … のように函数でない値も当然束縛できるし，またパターンも let {x, _} = pair in … のように使えるが，val は函数しか束縛できない[*46]．（相互）再帰函数を定義したい場合は val rec f_1(…) = e_1 and … and f_n(…) = e_n のように記述する．let の場合も同様．

- 前述の通り函数は Curry 化せず部分適用を許さないため，函数の型は一般的な τ' -> τ などではなく fun(τ_1, …, τ_n) -> τ の形式とし，例えば fun(τ_1, τ_2) -> τ は fun(τ_1) -> fun(τ_2) -> τ とは異なる型である．適用も例えば List.foldl f i xs ではなく List.foldl(f, i, xs) などと書く．やはり $e(e_1, e_2)$ と $e(e_1)(e_2)$ とは具象構文としてだけでなく抽象構文として異なる．

- 具象構文上型コンストラクタは前置で，例えば list<int> と記述する．複数の型引数がある場合は result<int, error> のようにコンマ区切り．高階の型コンストラクタはサポートしていない．

- 型註釈を書かずとも let 多相での主要型を推論してくれる．再帰函数については，型註釈を全ての引数と戻り値について書いていれば多相再帰も受理する．

- 直積型は {τ_1, …, τ_n} の形で書き，組（タプル）は Erlang と同様に {e_1, …, e_n} の形で書く．1 要素だけの直積型 {τ} と組 {e} もいくつかの用途のために用意されている．

- リストの式は [e_1, …, e_n] とコンマ区切りで書くほか，通常のいわゆるコンスセルの書き方 e_1 :: e_2 :: … :: e_n :: [] も可能．パターンでも同様．

- 函数定義や函数抽象の引数では，変数名だけでなく一般のパターンを使うことができる．

- 型変数は $a のようにドル記号が接頭辞としてつく．全称量化は <$a, $b> のようにコンマ区切りで型変数を並べて < と > で括り，束縛される函数名の直後に書く．シグネチャ中の宣言だけでなく，val have_same_length<$a, $b>(…) = … のように実装中の型註釈でもそのように書くことができる．

---

46 これは後述しますが Erlang のモジュール中では函数しか定義できないことに由来します．

4.2. 代数的データ型とモジュールによる抽象化

これも以下の例でほぼ伝わるかと思います：

```
module BinTree :> sig
  type t :: (o) -> o
  val leaf<$a> : fun($a) -> t<$a>
  val node<$a> : fun($a, t<$a>, t<$a>) -> t<$a>
  val size<$a> : fun(t<$a>) -> int
end = struct
  type t<$a> =
    | Empty
    | Node($a, t<$a>, t<$a>)

  val leaf(x) =
    Node(x, Empty, Empty)

  val node(x, tree1, tree2) =
    Node(x, tree1, tree2)

  val rec size(tree) =
    case tree of
    | Empty                  -> 0
    | Node(_, tree1, tree2) -> 1 + size(tree1) + size(tree2)
    end
end
```

`type t :: (o) -> o` が抽象化された型コンストラクタの宣言で，`(o) -> o` は 1 個の型引数をとることを表す種（カインド）です．2 個の場合は `(o, o) -> o` という具合にコンマ区切りで増えます．高階の種はサポートしていません．

4.3. 列多相によるレコード

レコードとは“ラベルつきの組”であり，Sesterl では $\{l_1 = e_1, \ldots, l_n = e_n\}$ の形で構築でき，$e.l$ で射影して成分を取り出せます． 簡単な例としては `{name = "Taro Tanaka", address = "Tokyo", age = 30}` のような具合です． また， レコード e に対して一部のラベルの値だけを上

書きしたレコードも $\{e \texttt{ with } l_1 = e_1, \ldots, l_n = e_n\}$ で構築できます．

レコードで非自明なのは，射影や更新にどんな型をつけるかです．例えば以下の `get_foo` にはどんな型がつくべきでしょうか？

```
val get_foo(r) = r.foo
```

例えば OCaml の場合だと，ラベルは各々 1 つのレコード用の型に属するものとして扱われるため，`get_foo` に相当する函数は `foo` というラベルをもつと定義されたただひとつのレコード型のみに対して定義された函数として単相になります[*47]．一方，ラベルが特定の型に紐づくものとは扱われない言語設計も当然ありえて，上記のような函数に “`foo` というラベルをもつレコードはなんでも受け取れて，その `foo` の型を戻り値の型とする” ということを表す多相な型をつけたりすることも十分考えられます．問題はその多相性をどう定式化するかで，既存の体系だと特に 2 つの方法がよく知られています：

- **SML#** 方式のレコード多相 [17]： 式・型・種の 3 段階からなる 2 階の型システムで，例えば `get_foo` 相当の函数には $\forall\alpha :: \mathrm{U}.\, \forall\beta :: \{\{\texttt{foo} : \alpha\}\}.\, \beta \to \alpha$ という具合の型がつきます．ここで U は任意の型につく種，$\{\{\texttt{foo} : \alpha\}\}$ は “`foo` というラベルをもち，それに対応する型が α であるようなレコード型” 全体につく種です．この型システムの特徴は，レコードに対して多相に型つけできるだけでなく，拡張や結合など一部のレコードに対する操作ができないという制約を設けつつも実行時のオーバーヘッドを減らせるような定式化になっていることです．
- **列多相** [7]： いろいろな変種がありますが，概して**列** (row) という構文的対象が型とは別にあるのが特徴です．列とは大雑把に言えば “レコードのラベルと型の組み合わせの一部だけ切り出したもの” で，例えば $\{\texttt{foo} : \texttt{int}, \texttt{bar} : \texttt{bool}, \texttt{baz} : \texttt{string}\}$ の “$\texttt{bar} : \texttt{bool}, \texttt{baz} : \texttt{string}$ の部分” といったものが列に該当します．2 階の型システムとして列多相を定式化すると，`get_foo` 相当の函数には $\forall\alpha :: \mathrm{Type}.\, \forall\rho :: \mathrm{Row}\{\texttt{foo}\}.\, \mathrm{record}\{\texttt{foo} : \alpha \mid \rho\} \to \alpha$ などという型がつく具合になります．ここで α は通常の型変数で，Type という種がつきますが，ρ は

47 OCaml では，例えば以下のようにレコード型 `person` を定義すると，`name`，`address`，`age` といったラベル名はそのスコープに於いて `person` 型専用のラベルになります：

```
type person = {name : string; address : string; age : int}
```

列変数と呼ばれ，列が全称量化されたものです．列 r に対して record$r$ が “r を内容としてもつレコード型” であり，$\{l : \tau | r\}$ とは “$l : \tau$ を列 r の先頭につけ足してできる列” です．レコードに含まれるものとして意味を為すために 1 つの列に於いては各ラベルは高々 1 回までしか出現してはいけないため，$\{l : \tau | r\}$ に於いて r はラベル l を含んでいてはならず，したがってここでの $\{\text{foo} : \alpha | \rho\}$ が意味を為すためには列変数 ρ は foo を含む列に instantiate されてはいけないことになります．こうした制約を表現するために列の種は Row$L$ という具合にラベルの集合 L を伴っており，ここでは L に相当するのが {foo} という 1 元集合というわけです．

Sesterl では v0.1.x の頃には SML# のレコード多相を導入していました[*48]が，次第に以下のような点が気になってきました：

- レコードに関してパフォーマンスがクリティカルに必要なわけではないので，実行時のオーバーヘッドを減らすようなコンパイル処理を実装するためにこの型システムを活かしたりはしていないし，それならばレコードに対する操作を制約している意味があまりない．
- 種が型に構文的に依存するため，以下のようなことが問題になる：
  - 通常の let 多相と違い型変数の全称量化の順序に意味があるなど，そもそも型システムとして幾分か複雑なため，型推論上の型と種の扱い方も複雑になりやすい．量化の順番に意味があるというのは，例えば上記の `get_foo` の型つけの場合に量化の順序を入れ替えて $\forall\beta :: \{\{\texttt{foo} : \alpha\}\}.\forall\alpha :: \mathrm{U}.\ \beta \to \alpha$ とすると意味を為さない別の型になってしまうことなど．
  - 他の型システムの拡張や型推論処理を内部的に効率化したアルゴリズムなどとこのレコードに関する推論が適切に両立できるかが非自明．

そこでこれを緩和する方法として Sesterl v0.2.0 からはレコード型の定式化を列多相に切り替えることにしました．上記の `get_foo` に対しては以下のように型がつきます：

```
val get_foo<$a, ?$r :: (foo)> : fun({foo : $a | ?$r}) -> $a
```

列変数は `?$` という接頭辞をもつことにし，列変数に対する種は (L) の形で記述することにし

48 これは何か列多相と比較の上決めたわけではなく，単に開発開始当時に筆者が SML# のレコード多相くらいしかレコードを多相的に扱う手法の直観を獲得していなかったことに起因します．

ました（複数のラベルからなる集合の場合はコンマ区切りで書きます）．また，単に `{foo : $a | ?$r}` と書いただけでレコード型として扱い，列自体を書く方法は（今のところ必要に感じていないため）ひとまず提供しないことにしました．

列多相に切り替えた際，型推論アルゴリズムの実装をかなり簡略化できたほか，既存のレコード多相に準拠して既に書いていたプログラムは型註釈を除いて書き換える必要なく移行でき，使い勝手が良くなりました．というわけで現在の Sesterl では列多相に基づいたレコードの型つけが採用されています．

4.4. ラベルつき引数

さて，引数が多くなっても何の引数なのかわかりやすくするために，かつ典型的な値が決まっている引数は適宜省略できるように，ラベルつき引数を導入したいのでした．まず，ラベルつき必須引数は “単にラベルを辞書順で並べて通常の引数に帰着する” という具合の方法で比較的簡単に実現できます．というわけで以下のように `-foo` というラベルによって必須引数を書けるようにしました：

```
val rec foldl(-f f, -init init, -list xs) =
  case xs of
  | []      -> init
  | y :: ys -> foldl(-init f(init, y), -list ys, -f f)
  end
```

再帰の適用で引数の順序が定義の際の引数の順序と違っていますがこれでも問題なく通るという算段です．この `foldl` には以下のように型がつきます：

```
val foldl<$a, $b> : fun(-f fun($a, $b) -> $a, -init $a, -list list<$b>) -> $a
```

というわけでラベルつき必須引数のサポートは簡単です．

一方でラベルつきオプション引数は結構非自明な点があります．とりあえず，必須引数では `-foo` だったラベルの形式をオプション引数では `?foo` にして，適用時に省略されたら `None` が渡されたことに，`?foo` v のように値が与えられていたら `Some(`v`)` が渡されたことにすることを考えましょう．ラベルつきオプション引数をもつ函数を定義する側はこれだけで OK です．例え

ば以下のように使えます：

```
val succ(n : int, ?diff diff_opt : option<int>) : int =
  case diff_opt of
  | None       -> n + 1
  | Some(diff) -> n + diff
  end
```

しかし，次のような高階函数にはどんな型をつけるべきでしょうか？

```
val use_optional(f) = f(42, ?foo 57) + 1
```

特にこの函数の引数 `f` にはどんな型がつくべきでしょうか？　`fun(int, ?foo int) -> int` のような型を想定するかもしれませんが，“`?foo` のほかに `?bar` のようなオプション引数も受け取れて，ここでは単に `?foo` しか与えられないだけ” という函数も `f` として適格なため，このことを表現する多相な型をつけたくなります．

実は，前節で触れた列多相をこのようなオプション引数の型つけに転用することができます．これは同様の仕組みを取っている論文や実装を見たことはなく，筆者が考案したつもりのものですが，仕組みとしてはシンプルであって比較的すぐ思いつくようなものです[*49]．

直観としては 1 つの函数のとるオプション引数全体を列だとみなすという方法です．すなわち，（簡単のためラベルつき必須引数は通常の引数と同一視して扱うこととして）函数の型は $\texttt{fun}(\tau_1, \ldots, \tau_n, r) \texttt{ -> } \tau$ という具合に列 r を 1 つオプション引数用にもっていると考えます．すると，上記の `use_optional` は $\forall \rho :: \mathrm{Row}\{\texttt{foo}\}.\, \texttt{fun(fun(int, \{foo : int} \mid \rho\}) -> int) -> int}$ という型をつければよさそうだ，という気がしてきます．こうして餘分なオプション引数をもつ函数でも列変数で吸収して扱えるようになります．Sesterl の具象構文としては以下のように記述することにしました：

```
val use_optional<?$r :: (foo)> : fun(fun(int, ?foo int, ?$r) -> int) -> int
```

49 多分大丈夫だろうとは信じているものの，この手法の正当性（保存や進行といった型安全性）を証明したというわけではないです．論文にしたためるほどの内容とは思いませんが，餘裕があるときに証明してみたいことではあります．

4.5. FFI

FFI は，特に Erlang 製の既存実装を再利用しやすくしたり，Erlang 製の実装から Sesterl 製の実装への段階的な移行を容易にしたりするために，是非とも簡潔なものにしたい機能でした．以下のような記述で FFI が実現できます：

```
val binary_to_chars : fun(binary) -> list<char> = external 1 ```
  binary_to_chars(Bin) ->
      erlang:binary_to_list(Bin).
```
```

すなわち，文字列リテラル[*50]で Erlang の函数定義が直接書け，それに型をつけて Sesterl 側から使うことができます．`external` の直後にはアリティを指定します[*51]．概要だけ説明すると，Sesterl の値と Erlang の値との紐づけは以下のようになっています：

- 整数，浮動小数点数，バイナリ，PID，リスト，タプルなどの値はそのまま自然な形で Erlang の値になっている．
- レコードは，ラベルをアトムのキーとするマップで表現される．例えば Sesterl でのレコード `{name = "Taro Tanaka", age = 30}` は Erlang でのマップ `#{name => <<"Taro Tanaka">>, age => 30}` へとコンパイルされ，FFI ではマップとして Sesterl のレコードを操作できる．
- 代数的データ型の値に関しては，引数を取らないコンストラクタ $C$ は小文字始まりのアトム $c$ で，引数を取るコンストラクタ $C(v_1, \ldots, v_n)$ はアトムを先頭とするタプル $\{c, v_1, \ldots, v_n\}$ で表現される．$C$ と $c$ の対応は例えばアッパーキャメルケースの `SomeConstructor` がスネークケースの `some_constructor` に変換される要領．例えば Sesterl での `Node(42, Node(57, Empty, Empty), Empty)` は Erlang での `{node, 42, {node, 57, empty, empty}, empty}` へとコンパイルされる．ただし，`option` 型の `Some` と `None` は利便性のため例外的にそれぞれ

50 Sesterl では `"hello"` のような通常の二重引用符や `` `hello` `` という Markdown に似た形式の文字列リテラルがあり，通常の文脈で書くと Erlang のバイナリ `<<"hello">>` になります．FFI の構文ではこの文字列リテラルの形式を “Erlang コードを書くための領域” として転用しています．

51 このアリティは Sesterl コンパイラが `-export` の内容を出力するのに必要です．文字列リテラル内の Erlang のコードを構文解析するようにすればアリティを書く必要はないのですが，現状では Erlang コードは構文解析せず出力する Erlang ソースコードにそのまま貼りつける仕組みになっています．
```

ok と error で表現される[*52].

- 函数定義および函数抽象では，通常の引数とラベルつき必須引数は Erlang の普通の引数として扱われ，ラベルつき必須引数はラベルの辞書順で並ぶ．オプション引数は最後の 1 引数として単一のマップになっており，オプション引数をひとつも与えない場合はその最後の 1 引数が欠ける． そのため, FFI の函数でオプション引数を受け取るものにしたい場合は Sesterl 側では 1 つの函数であっても Erlang 側ではオプション引数のマップを受け取るものと受け取らないものの 2 種を与える必要がある．

4.6. 並行処理

今のところ Sesterl の実際のユースケースではどちらかといえば OTP ライブラリを使用するのが中心で, “生の Erlang の機能としてのプロセス生成や送受信” は結果的にあまり使う必要がなさそうな様相です．しかし，専ら OTP ライブラリを使う場合に於いても並行処理の型つけの仕組みは重要なので，どんな風に型つけを行なうことにしたか紹介したいと思います．

Sesterl でも，Erlang の self，spawn，!，receive はほぼそのままの意味論で提供しています．Erlang の self，spawn，! に相当する処理はそれぞれ組み込みの self，spawn，send という函数で使えるようになっており， またこれに加えて return という （モナドの return 或いは pure に相当する）純粋な計算結果を並行処理の結果に持ち上げる函数があります．結論から掲げると，それぞれ以下のような型がついています：

```
val self<$a> : fun() -> [$a]pid<$a>
val spawn<$a, $b> : fun(fun() -> [$b]unit) -> [$a]pid<$b>
val send<$a, $b> : fun(pid<$b>, $b) -> [$a]unit
val return<$a, $b> : fun($b) -> [$a]$b
```

fun(τ_1, …, τ_n) -> [τ']τ および pid<τ> という見慣れない形式の型がありますが，まず前者が並行計算につける型です．気持ちとしては [τ']τ の部分がモナディックな機構に相当し，おおよそ “τ' 型のメッセージを受け取れて，最終的に τ 型の値になるような並行処理” を表します[*53].

52 ユーザ定義の代数的データ型でも，本来の規則とは違うアトムにコンパイルしたい場合はそのように指定する註釈がつけられるようにしています．

53 「τ' 型のメッセージを受け取れる」は，より正確には「メッセージを受け取るなら，そのメッセージは τ' 型でなければならない」です．

また，後者の pid<τ> が “τ 型のメッセージを受け取れるプロセスの PID につく型” です．

もう少しボトムアップかつフォーマルに書くと，（抜粋ですが）型と式の構文は以下のように設定されています：

$$
\begin{aligned}
\tau &::= \texttt{fun}(\tau,\ldots,\tau)\ \texttt{->}\ \tau \mid \cdots \\
&\quad \mid \texttt{fun}(\tau,\ldots,\tau)\ \texttt{->}\ [\tau]\tau \mid \texttt{pid<}\tau\texttt{>} \\
e &::= e(e,\ldots,e) \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \cdots \\
&\quad \mid x \mid \texttt{fun}(x,\ldots,x)\ \texttt{->}\ e\ \texttt{end} \mid \texttt{fun}(x,\ldots,x)\ \texttt{->}\ \texttt{act}\ k\ \texttt{end} \\
k &::= e(e,\ldots,e) \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ k \mid \texttt{if}\ e\ \texttt{then}\ k\ \texttt{else}\ k \mid \cdots \\
&\quad \mid \texttt{do}\ x\ \texttt{<-}\ k\ \texttt{in}\ k \mid \left(\texttt{receive}\ p\ \texttt{->}\ k \mid \cdots \mid p\ \texttt{->}\ k\ \texttt{end}\right)
\end{aligned}
$$

式が純粋計算用の e と並行計算用の k に分かれているのが特徴です． モナドの bind に相当するのが $\texttt{do}\ x\ \texttt{<-}\ k_1\ \texttt{in}\ k_2$ という構文で[*54]，これにより送受信に関わる処理をシーケンシャルに繋ぎます．receive ⋯ end は Erlang の receive 式にほぼそのまま対応するものです．p は具体的な定義は省略しますがパターンを表すメタ変数です．$\texttt{fun}(x_1,\ldots,x_n)\ \texttt{->}\ \texttt{act}\ k\ \texttt{end}$ が並行処理をなすような函数抽象で，以下の型つけ規則でみるようにこの函数は並行処理の式に於いてしか適用できませんが，この函数自体は純粋な値と扱います．

型判定は， 純粋な式に対しては $\Gamma \vdash e : \tau$ の 3 項関係の形を， 並行処理の式に対しては $\Gamma \vdash k : [\tau']\,\tau$ の 4 項関係の形をそれぞれとり，以下のような規則で型をつけます：

$$\frac{\Gamma \vdash k_1 : [\tau']\,\tau_1 \qquad \Gamma + \{x \mapsto \tau_1\} \vdash k_2 : [\tau']\,\tau_2}{\Gamma \vdash \texttt{do}\ x\ \texttt{<-}\ k_1\ \texttt{in}\ k_2 : [\tau']\,\tau_2}$$

$$\frac{\vdash p_i : \tau' \Rightarrow \Gamma_i \qquad \Gamma + \Gamma_i \vdash k_i : [\tau']\,\tau\ (\text{for each } i \in \{1,\ldots,n\})}{\Gamma \vdash \texttt{receive}\ p_1\ \texttt{->}\ k_1 \mid \cdots \mid p_n\ \texttt{->}\ k_n\ \texttt{end} : [\tau']\,\tau}$$

$$\frac{\Gamma + \{x_1 \mapsto \tau_1,\ldots,x_n \mapsto \tau_n\} \vdash k : [\tau']\,\tau}{\Gamma \vdash \texttt{fun}(x_1,\ldots,x_n)\ \texttt{->}\ \texttt{act}\ k\ \texttt{end} : \texttt{fun}(\tau_1,\ldots,\tau_n)\ \texttt{->}\ [\tau']\tau}$$

ただし，型環境の結合 $\Gamma_1 + \Gamma_2$ は “変数名が重複した場合は右を優先する合併” であり，またパターンに対する判定 $\vdash p : \tau \Rightarrow \Gamma$ は “p は τ 型の値にマッチしうるパターンであり，その中に出現する変数に対する型環境は Γ” であることに対応する判定です（導出規則は素直なので省略します）．$\texttt{do}\ x\ \texttt{<-}\ k_1\ \texttt{in}\ k_2$ の規則はすなわち変数 x が k_1 の計算結果に束縛され，また k_1 と k_2 は同一の形式のメッセージを受け取る並行処理でなければならないという制約を表します．receive

54 実際には変数の部分はパターンに一般化することができ，$\texttt{do}\ p\ \texttt{<-}\ k_1\ \texttt{in}\ k_2$ の構文をとりますが，ここでは簡単のため変数のみに限定しています．

p_1 -> k_1 | ⋯ | p_n -> k_n end に対する型つけ規則は各 p_i が現在の文脈で受信できるメッセージの型である τ' 型の値にマッチするパターンであること，かつ続きの並行処理である各 k_i でもやはり受信できるメッセージの型は引き続き τ' であることを要請します．

函数適用は純粋 / 並行でそれぞれ以下のように型つけされます．純粋函数は純粋な式の中，つまり e が期待される箇所でのみ適用できるようになっており，一方で並行処理を生じる函数は並行処理の式の中，つまり k が期待される箇所でしか適用できないようになっています：

$$\frac{\Gamma \vdash e : \texttt{fun}(\tau_1, \ldots, \tau_n) \texttt{ -> } \tau \qquad \Gamma \vdash e_i : \tau_i \text{ (for each } i \in \{1, \ldots, n\})}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{fun}(\tau_1, \ldots, \tau_n) \texttt{ -> } [\tau']\tau \qquad \Gamma \vdash e_i : \tau_i \text{ (for each } i \in \{1, \ldots, n\})}{\Gamma \vdash e(e_1, \ldots, e_n) : [\tau']\,\tau}$$

適用の規則をこの節の冒頭で掲げた `self`, `spawn`, `send`, `return` の型と組み合わせて許容規則をつくると以下のようになります（というより，以下のような規則を着想してそこから逆算すると冒頭のような多相型がつくというわけです）：

$$\frac{}{\Gamma \vdash \texttt{self()} : [\tau']\,\texttt{pid<}\tau\texttt{>}} \qquad \frac{\Gamma \vdash e : \texttt{fun() -> [}\tau\texttt{]unit}}{\Gamma \vdash \texttt{spawn}(e) : [\tau']\,\texttt{pid<}\tau\texttt{>}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{pid<}\tau\texttt{>} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{send}(e_1, e_2) : [\tau']\,\texttt{unit}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) : [\tau']\,\tau}$$

ここまでくるとわりと直観的な型つけ規則に見えてくるんじゃなかろうかと思いますが，一応直観を書くと以下のような具合です：

- `self` は，受信できるメッセージの型が τ' であるような文脈で使うと，戻り値である自身の PID は当然 τ' 型のメッセージを受け取れるプロセスの PID なので，戻り値は `pid<`τ'`>` 型．
- `spawn` は，τ 型のメッセージを受け取る並行処理の函数抽象 e を受け取ると，τ 型のメッセージを受け取れるプロセスを生成してその PID を返すので，戻り値は `pid<`τ`>` 型．また，どんなメッセージが受信できる文脈で呼び出さねばならないかの制約はないので，τ' は自由．
- `send` の適用では，第 1 引数の PID のプロセスが受け取れる型と，第 2 引数のメッセージとなる式の型が一致していなければならない．
- `return` は本当にそのまま．τ' は自由．

実際にここまでに述べた型つけ規則で，2.2 節の Erlang の解説で掲げた例を Sesterl で型のつくプログラムへとほぼそのままの形で移植することができます：

```
module ConcurrencyExample = struct
  val sub() = act
    receive
    | {from, n} ->
        let answer = ModuleB.calculation_b(n) in
        send(from, answer)
    end

  val main() = act
    do pid_a <- self() in
    do pid_b <- spawn(fun() -> act sub() end) in
    do {} <- send(pid_b, {pid_a, 42}) in
    let result_a = ModuleA.calculation_a(57) in
    receive
    | result_b -> return({result_a, result_b})
    end
end
```

並行処理を行なうような函数をトップレベルで定義する場合は `val` $f(x_1, \ldots, x_n)$ = `act` k `end` の形で記述します．なお，上記の例でもし `ModuleB.calculation_b` が内部で並行処理を行なう函数だったなら以下のように書く必要があります（`ModuleA.calculation_a` も同様）：

```
do answer <- ModuleB.calculation_b(n) in
```

ちなみにこの節で述べたような並行処理に対するモナディックな型つけはしばしば使われるテクニックで，似た直観に基づく型システムが例えば λ_{act} [6] や FPCF と session calculus [18] にみられるほか，Elm の `Process` モジュール [2] もよく似た型つけを採用しています．

4.7. 並行処理に関する型がなぜ函数型と一体化したような形になっているのか

前節で紹介した並行処理の型に於いては“$[\tau']\tau$ の部分だけを単独で取り出すことができない”ように函数の終域のみに制約しており，かつそのために式の構文も純粋な e と並行処理を担

う k に分離していました．このような定式化になっているのは，端的に Erlang へのコンパイルの都合です．もしこの $[\tau']\tau$ の部分だけで型をなすとすると，この型がつく式はどのようにコンパイルすればよいでしょうか？　例えば次のようなプログラムを考えます：

```
val f(pid) =
  let _ = send(pid, "Hello") in
  42
```

この函数 f には fun(pid<binary>) -> int がつくとするのが自然でしょう．すなわち，f は少なくとも型だけ見れば純粋函数です．したがって，send によって実際に送信が発生するのではない意味論にしなければ，純粋な計算と並行処理を区別するモナディックな型をつけた意義がありません．これは端的に言えば Haskell で次のような函数が純粋函数であるのと同様です：

```
f :: String -> Int
f s =
  let _ = putStrLn s in
  42
```

元々の例は，要するに“pid に "Hello" を送信する並行処理”だけを作成し，その並行処理を実際に動作させずに捨てているというわけです．したがって，send の適用の戻り値である $[\tau']$unit 型のつく値は“未実行の並行処理”なのであって，Erlang にコンパイルするには（アリティ 0 の）函数抽象にする必要があります．また，これに伴って，例えば do x <- k_1 in k_2 という式は，x に対応して生成された Erlang での変数名を X，同様に k_1 のコンパイル結果を K_1，k_2 のコンパイル結果を K_2 として，fun() -> X = K_1(), K_2() end のような式にコンパイルされねばなりません．こうした対処をしていくと，かなり多くの処理が fun() -> … end という函数抽象による遅延と空の引数列の適用を繰り返すことになり[*55]，パフォーマンスの悪化が見込まれます．或る程度最適化で緩和できるようにも思えますが，とりわけトップレベルで定義される並行処理を含む函数のパフォーマンスを改善することが難しく，できたとしても FFI をわかりやすくするという目的のひとつを達成できるとは考えがたいです．

そこで，前節で紹介したように並行処理のモナドを最初から函数と一体化したものとして定式化し，構文も純粋な式と並行処理の式とで分けるという方式を採ることにしました．こ

55 ここでの函数抽象による遅延機構と空の引数列の適用は，しばしばそれぞれ thunk と force と呼ばれます．

れにより fun(τ_1, …, τ_n) -> τ 型の式 （＝引数を受け取って純粋な計算を行なう函数） と fun(τ_1, …, τ_n) -> [τ']τ 型の式（＝引数を受け取って並行処理を行なう函数）は全く同じ式へとコンパイルできるようになります．トップレベルの函数にも純粋 / 並行の違いで特別な区別を施す必要がないなど FFI としてもわかりやすいほか，無駄な遅延機構でパフォーマンスを損なったりもしにくいものになりました．こうした定式化はほかの AltErlang で行なわれていてもおかしくないとは思いますが，今のところ知る限り Sesterl 独自の方法です．

4.8. ファンクタによる OTP ライブラリの定式化

さて，OTP ライブラリを（OTP Design Principles に従う形で）自然に使えるような言語設計にすることが Sesterl の要件のひとつでした．これについて取り組んだ研究は筆者の感知する限りあまり見かけず，また実際に実装された AltErlang の言語設計やライブラリなどを見てもそうそう見当たらないものでした[*56]．要するに環境要因としては “それなりに腕の鳴る感じの” 課題だったわけです．

で，Sesterl でどのように gen_server や supervisor といった OTP ライブラリを（それ自体には手を加えたり再実装したりせずに）ラップして型安全に使えるように型のつくインターフェイスにしたかというと，別にアイディアとして見事なものというほどのことはありません．簡単に言えば，OTP ライブラリのインターフェイスを ML 系言語が備えているファンクタの機構によって定式化したということになります．より具体的に言えば，次のような直観による定式化です：

> init や handle_call などのコールバック函数を要求する gen_server ビヘイビアをシグネチャと捉えると，gen_server モジュールは，このシグネチャのつくストラクチャを受け取ってプロセスの実装となるストラクチャを返すようなファンクタとして定式化できる．

ただし，実際にはコールバック函数に相当する函数だけでなくそれらの函数にまつわる型の指定もファンクタの引数として与えられねばなりません．結果として gen_server ビヘイビアに対応するシグネチャ Stdlib.GenServer.Behaviour [*57]は以下のような定義になりました（一部省略）：

---

56 少なくとも開発開始当時の筆者は認知していませんでした．現在だと 3.3 節で紹介したように Hamler が型クラスとしてビヘイビアを定式化しているようです．

57 綴り字が英国英語になっているのはなんとなくですが，米国英語に変えてもいいかもしれません．

```
module GenServer : sig
  ...
  type initialized :: (o) -> o
  type reply :: (o, o) -> o
  type no_reply :: (o) -> o

  signature Behaviour = sig
    type init_arg :: o
    type request :: o
    type response :: o
    type cast_message :: o
    type info :: o
    type state :: o
    ...
    val init : fun(init_arg) -> [info]initialized<state>
    val handle_call<$a> :
      fun(request, pid<$a>, state) -> [info]reply<response, state>
    val handle_cast :
      fun(cast_message, state) -> [info]no_reply<state>
    val handle_info : fun(info, state) -> [info]no_reply<state>
    val terminate : fun(StopReason.t, state) -> [info]unit
    ...
  end
  ...
end
```

initialized, reply, no_reply は抽象化された型ですが， それぞれ単に init, handle_call, handle_cast（および handle_info）の戻り値のための型で，それぞれ Erlang の gen_server での {ok, State} | …, {reply, Reply, State} | …, {noreply, State} | … の形式に対応するものです． これらの型のつく値は GenServer モジュールが提供する以下のような函数によってつくることができます[58]：

```
module GenServer : sig
```

58 説明の都合で実際についている型より少し特殊化しています．

```
  ...
  val init_ok<$m, $state> : fun($state) -> [$m]initialized<$state>
  val init_stop<$m, $state> : fun(StopReason.t) -> [$m]initialized<$state>
  val reply<$m, $response, $state> :
    fun($response, $state) -> [$m]reply<$response, $state>
  val reply_and_stop<$m, $response, $state> :
    fun(StopReason.t, $response, $state) -> [$m]reply<$response, $state>
  ...
end
```

これらの函数は実際に Stdlib.GenServer.Behaviour シグネチャをつけるストラクチャのメンバである init, handle_call, handle_cast, … の定義をする際に使います.

Stdlib.GenServer.Behaviour が要求するそれぞれの型の用途は以下の通りです：

- init_arg： init の引数.
- request： 返信を要請する同期的メッセージの型.
- response： 返信のメッセージの型.
- cast_message： 返信を要請しない非同期的メッセージの型.
- info： handle_info で捌かれるより一般の非同期的メッセージの型.
- state： プロセスが保持する状態の型.

そして，こうしたシグネチャ Stdlib.GenServer.Behaviour のつくストラクチャを引数にとってプロセスで稼働する実装のストラクチャを返すのが，次のようなシグネチャをもつファンクタ Stdlib.GenServer.Make です：

```
module GenServer : sig
  ...
  type start_link_error = RawValue.t
  type call_error = RawValue.t

  module Make : fun(C : Behaviour) -> sig
    type proc :: o
    val as_pid : fun(proc) -> pid<C.info>
    val from_pid : fun(pid<C.info>) -> proc
    val start_link<$a> :
```

```
      fun(C.init_arg) -> [$a]result<proc, start_link_error>
    val call<$a> :
      fun(proc, C.request, ?timeout int) ->
        [$a]result<C.response, call_error>
    val cast<$a> : fun(proc, C.cast_message) -> [$a]unit
    val send_info<$a> : fun(proc, C.info) -> [$a]unit
    ...
  end
  ...
end
```

Stdlib.GenServer.Bahaviour シグネチャのつく C というモジュール引数を受け取り，そのメンバを使って戻り値のモジュールを構成します．戻り値のモジュールには start_link や call などの函数のメンバがあり，これが元々の OTP での gen_server:start_link(?MODULE, …) や gen_server:call として使えるという算段です．proc は戻り値のモジュールによってつくられるプロセスの PID の型であり，その気になれば as_pid と from_pid によって通常の PID と相互に変換することができます[*59]．

シグネチャだけ記載しても具体的な使用例が想像しにくいと思うので，2.3 節のセルプロセスの例を Stdlib.GenServer.Make を使って Sesterl に移植してみます：

```
module Cell :> sig
  open Stdlib

  type proc
  type start_error = RawValue.t
  val start_link<$a> : fun(int) -> [$a]result<proc, start_error>
  val get<$a> : fun(proc) -> [$a]option<int>
  val set<$a> : fun(proc, int) -> [$a]unit
  val stop<$a> : fun(proc) -> [$a]unit
end = struct
  open Stdlib
```

59 proc は単に pid<C.info> 型のエイリアスにしてもよいのですが，proc 型の値が意識しないうちに通常の PID として使われ，他の形式の（たまたま受信できるメッセージの型が同じ）プロセスに由来する PID と混同されるのを防ぐために意識的に相互変換しなければならない API にしました．

```
type start_error = RawValue.t

module Callback = struct
  type request =
    | Get

  type response =
    | Got(int)

  type cast_message =
    | Set(int)
    | Stop

  type state = int
  type init_arg = int
  type info = unit
  ...
  val init(n) = act
    GenServer.init_ok(n)

  val handle_call(req, pid, n) = act
    case req of
    | Get -> GenServer.reply(Got(n), n)
    end

  val handle_cast(msg, n) = act
    case msg of
    | Set(m) -> GenServer.no_reply(m)
    | Stop   -> GenServer.no_reply_and_stop(StopReason.normal(), n)
    end

  val handle_info(_, n) = act
    GenServer.no_reply(n)

  val terminate(_, _) = act
    return({})
```

```
    ...
  end

  module Core = GenServer.Make(Callback)

  type proc = Core.proc

  val start_link(n : int) = act
    Core.start_link(n)

  val get(pid : proc) = act
    do res <- Core.call(pid, Callback.Get) in
    case res of
    | Ok(Callback.Got(n)) -> return(Some(n))
    | Error(_)            -> return(None)
    end

  val set(pid : proc, m : int) = act
    Core.cast(pid, Callback.Set(m))

  val stop(pid : proc) = act
    Core.cast(pid, Callback.Stop)
end
```

Erlang と違って型の定義が必要な影響もあってコードがちょっと長いですが， やっていることは別に複雑ではありません． コールバック函数や型定義をメンバにもつ `Callback` という内部的なモジュールをつくり， それを `GenServer.Make` に渡して生成された戻り値のモジュールを `Core` と名づけ, `start_link` や `get` といった函数の定義に使っているという具合です.

というわけでこういった要領で `gen_server` に型をつけてラップすることができ, Sesterl から OTP ライブラリを型安全に使用することができるようになっています． 詳細に紹介すると `gen_server` 以上に長くなるので省きますが, `supervisor` も同様にしてファンクタで定式化することができています．やったぜ.

モジュールシステムにもいろいろな定式化がありますが, Sesterl では冒頭で述べた通り F-ing modules [22] に基づくものを採用しました．F-ing modules 自体については本記事では詳

しくは触れませんが，バックナンバーである『ヤバイテックトーキョー vol.5』で筆者が「F-ing modules の型検査とコンパイル手法」という記事を寄稿していますので，もし興味をお持ちでしたら眺めてみてください．

4.9. モジュールのコンパイル方法

さて，API としてはファンクタが OTP ライブラリに型をつけてラップするのに筋が良さそうな形式であることは前節で見ました．しかし，まだ問題が残っています： ファンクタないし一般のモジュールは，どのような Erlang プログラムにコンパイルすべきでしょうか？

前提的な方針としては，Sesterl での入れ子のない 1 ストラクチャは Erlang の 1 モジュールにコンパイルしたいと考えます．これは主に以下のような理由によります：

- FFI をわかりやすいものにしたいため．
- Erlang のソースファイルと Sesterl のソースファイルとをモジュール単位で切り分けて共存させられるようにしたいため．
- Erlang のランタイムには，**ホットコードローディング**という，プログラムを稼働させたままモジュール単位で実装を新しいものに置き替えるという驚きの機能があり，将来的にこれをなるべく自然な形でサポートしたいため．

ここから一般のモジュールのコンパイル方法へと拡張したいわけです．

まず， 単にストラクチャが入れ子になった構造にコンパイル方法を拡張するのは簡単です．単なる木構造なので，各ノードをモジュールにばらし，モジュール名はそれぞれその “パス” にするとよいだけです．例えば以下のような構造は，

```
module Foo = struct
  module Bar = struct
    val f(...) = ...
  end
  module Baz = struct
    val g(...) = ...
  end
  val h(...) = ...
end
```

以下のような 3 つのモジュールへとコンパイルすればよいです：

```
-module('Foo.Bar').
-export([f/...]).
f(...) -> ...
```

```
-module('Foo.Baz').
-export([g/...]).
g(...) -> ...
```

```
-module('Foo').
-export([h/...]).
h(...) -> ...
```

しかし，ファンクタはそう簡単にはいきません．Erlang ではそもそも動的にモジュールを生成する機能がないため，直接ファンクタ相当の機構を Erlang に落とし込む手段がないのです．一応これを解決する方法として「Erlang でファンクタがそのままでは表現できないことをもって上記のようなモジュールのコンパイル方法は諦めてしまい，Sesterl に於けるストラクチャはレコードと同様に，ファンクタは函数抽象と同様のものとして Erlang の式にコンパイルすることにする．モジュールはそれよりも大きい何らかの単位とする」ということも考えられます．ですが，これだと上記で述べた利点の裏返しで，Sesterl のモジュールの単位でホットコードローディングを行なうことはできなくなり，FFI がわかりにくいものになることも間違いなく，また直接 Erlang で書かれたプログラムと Sesterl で書かれたプログラムが共存させるのも厄介になりそうですから，やはりできればやりたくないものです．

そこで登場するのが「ファンクタの適用は静的に解決し，戻り値のモジュールをコンパイル時に生成してしまう」という考え方です．実際，このような考えには先行研究があり，**static interpretation** と呼ばれる手法が提案されています．そのひとつが **Futhark** という言語向けのもの [4] で，これは GPGPU 向けにコンパイルされる函数型の言語で実行時のオーバーヘッドを

減らすためにファンクタの適用やストラクチャの入れ子などをコンパイル時に除去することを意図して設計された体系で，ソース言語としては F-ing modules とよく似ています*60. Sesterl だとストラクチャの構造は残したいので全くそのまま実装すればよいわけではありませんが，ほぼこの体系の方針に従ってファンクタを含むプログラムをストラクチャの木構造のみにする過程を経て Erlang のモジュール群へとコンパイルすることができます.

static interpretation の詳細は述べるとかなり長くなるので，やはりバックナンバー『ヤバイテックトーキョー vol.5』の筆者の記事「F-ing modules の型検査とコンパイル手法」に投げたいと思います.

というわけで, Sesterl では static interpretation によってファンクタの適用を静的に解決してしまうことで OTP ライブラリなどファンクタを使って書かれたプログラムも Erlang モジュールへと落とし込めるようなコンパイル処理が実現できています．やったぜ.

4.10. 再帰モジュール（未実現）

前々節や前節で見たように F-ing modules と static interpretation に基づくファンクタを含むモジュールシステムは OTP ライブラリの型つきの定式化などに大いに役立っていますが，顕著に必要ながらまだ備えつけられていない機能があります．それが**再帰モジュール** (recursive modules) です.

再帰モジュールとはその名の通り （再帰函数が `let rec` で相互再帰的に定義されるように）相互再帰的に定義されたモジュールです． 例えば OCaml では（依然として実験的な言語機能であると警告されていますが）`module rec` $X_1 : S_1 = M_1$ `and` $\cdots$ `and` $X_n : S_n = M_n$ という構文で再帰モジュールを定義することができます.

再帰モジュールは OCaml がサポートしようとしているなど一応計算機言語一般にとってありがたみのある機能で，どんなプログラムが書けて嬉しいかなどは [13] にいくつか具体例がありますが， とはいえそれほど真に必要となる場面が多いわけではありません． しかしながら，Sesterl ではとりわけ再帰モジュールの機能が必要に迫られることが顕著に多いです． なぜかというと, Erlang では並行計算のために自然にモジュールが相互に依存しあう実装になることがよく見られるためです.

60 正確には Futhark のモジュールシステムの方が F-ing modules に比べていくらか制限を設けられています.

Sesterl に移植するなら再帰モジュールが必要になるような Erlang プログラムの簡単な例として, 2 つの gen_server ビヘイビアのコールバックモジュール foo_server と bar_server があり， これらのモジュールによる 2 つのプロセスがメッセージを送り合う状況を考えてみます． このとき， 常にどちらか片方が他方に対して通信を開始するのであれば特に問題はありません． もし foo_server によるプロセスと bar_server によるプロセスとの間の通信は常に foo_server 側から開始され, bar_server 側は受け取るだけだったり即座に返信するだけだったとすると, bar_server は例えば piyo という事象を伝えるメッセージをその bar_server の実装に基づいて動作するプロセスに対して送るための notify_piyo という函数を API として公開し, foo_server の実装中で bar_server:notify_piyo を呼ぶようにしておけば事足ります.

しかし， 問題はどちらも相手に対して能動的に通信を開始する状況がある場合です． この場合, bar_server が notify_piyo のような函数を API として提供してそれを foo_server の実装が呼ぶだけでなく, foo_server も notify_hoge のような函数を公開して bar_server の実装が foo_server:notify_hoge を呼ぶという形式をとるのが自然なカプセル化です． その結果, foo_server と bar_server はモジュールとして相互依存な実装になります. Erlang ではモジュール間の参照関係には特に制約がないためこのような相互依存な実装がごく普通に書けますが, こうした相互依存なモジュールを型つきの世界に持ってくる場合, 再帰モジュールの機構が要請されるわけです.

再帰モジュールを実現したモジュールシステムの既存研究はいろいろあり [3, 13, 21]， 再帰のないモジュールシステムに比べて顕著に実現が難しい[*61]ため三者三様に地獄の様相を呈しています． で, Sesterl にもそれを参考にして設計・実装するかと思うわけですが， 残念ながらそうは問屋が卸しません．というのも, Sesterl は既に述べたように static interpretation に依存してモジュールをコンパイルしているためです．再帰モジュールと static interpretation を共存させるのはかなり非自明で，そもそも static interpretation はファンクタ適用を静的に展開しても必ず停止することを前提しているため，素直に再帰モジュールに拡張しようとすると適用の展開が停止しない可能性が生じ，コンパイルを伴う型検査が決定不能になるおそれがあります．こうした点を克服するのはかなり骨が折れそうで，おそらく解決すればかなり良い論文になるのではないかと思います．まあそんな問題が簡単に解決できることはなく，それゆえに未実装どころか言語設計としても全くの未解決なのが現状です.

というわけで，総合的にはなかなか良い感じにモジュールシステムが実現できてきているも

61 とりわけ double vision problem [3, 13, 21] という言語設計上の問題への対処が厄介です.

のの，再帰モジュールは大きな課題として残っています．

5. 天九のルール

次章で実際に Sesterl（と Elm）を用いてオンラインゲームを設計・実装した話をする前に，ここではちょっと休憩して**天九**のルールを簡単に紹介します．正直なところ話の流れとしてはそんなに必須の内容ではなく，大体麻雀のように配牌が配られて 4 人が逐一出していくという手順を経る一般的な卓上ゲームであることさえ把握していれば次章が読めるのですが，そもそも天九のルールを記載した和文の文献が少ないので布教用として書いておきます．なかなか良いゲームで，比喩でなく 1 日中やっていても飽きないくらいのめり込めます．もし良ければここでルールを覚えて天九 Online で遊んでみてください．気に入ったら実際に**天九牌**を入手して物理的に遊んでみることもおすすめします[*62]．

5.1. 使用するコンポーネント

天九で使う**天九牌**は，図 2 に示すような 32 枚で 1 セットの牌です．32 枚は**文牌**（ウェンパイ）と**武牌**（ウーパイ）という 2 つの非対称なスートからなっており，文牌は 11 ランクが各 2 枚ずつの 22 枚，武牌は 6 ランクあってそのうち 4 ランクが 2 枚，2 ランクが 1 枚だけという構成の 10 枚です．武牌は（雑九の 2 枚が 4-5 と 3-6 であるなど）同ランクの 2 枚も互いに図柄が違っていますが，互いに全く同じ牌と扱われます．

牌の名称や発音はおそらく地域によって少しずつ異なるかと思いますが，カタカナで近似するとおおよそ次のように読まれます[*63]： 文牌は**天**（ティエン），**地**（ディ），**人**（レン），**和**（ハー），**梅花**（メイファ），**長三**（チャンサン），**板凳**（バンデン），**斧頭**（フートウ），**紅頭十**（ホントウシー），**銅錘六**（トンチュイリュー），**高脚七**（カオジャオチー），武牌は**雑九**（ザーチュー），**雑八**（ザーパー），**雑七**（ザーチー），**六**（リュー），**雑五**（ザーウー），**三**（サン）です．武牌は便宜的に単に九とか八とか数値で呼ぶことも多いです．

---

62 日本ではなかなか出回っていませんが，ゲームマーケットなどの即売会で販売している方がいたりするほか，オンラインで中華圏から個人輸入する手段もあります．筆者は主に輸入代行業者に依頼して入手し 6 セットくらい所有しています．いろんなメーカーのものがあるようですが，なるべく分厚い牌のものを選ぶとよいです．本体価格は結構手頃で 1000–2000 円程度だったりしますが，重量があるので手数料や運搬費がかさみ，1 セットで 4000–5000 円程度にはなります．また，重量に対する運搬費は概して y 切片が大きいので，同時にいくつも購入すると 1 セットあたりの出費は割安になったりします．

63 日本語圏で（おそらく）最初に天九牌を熱心に広めた方である伊藤拓馬さんによる音訳 [26] におおよそ従っています．

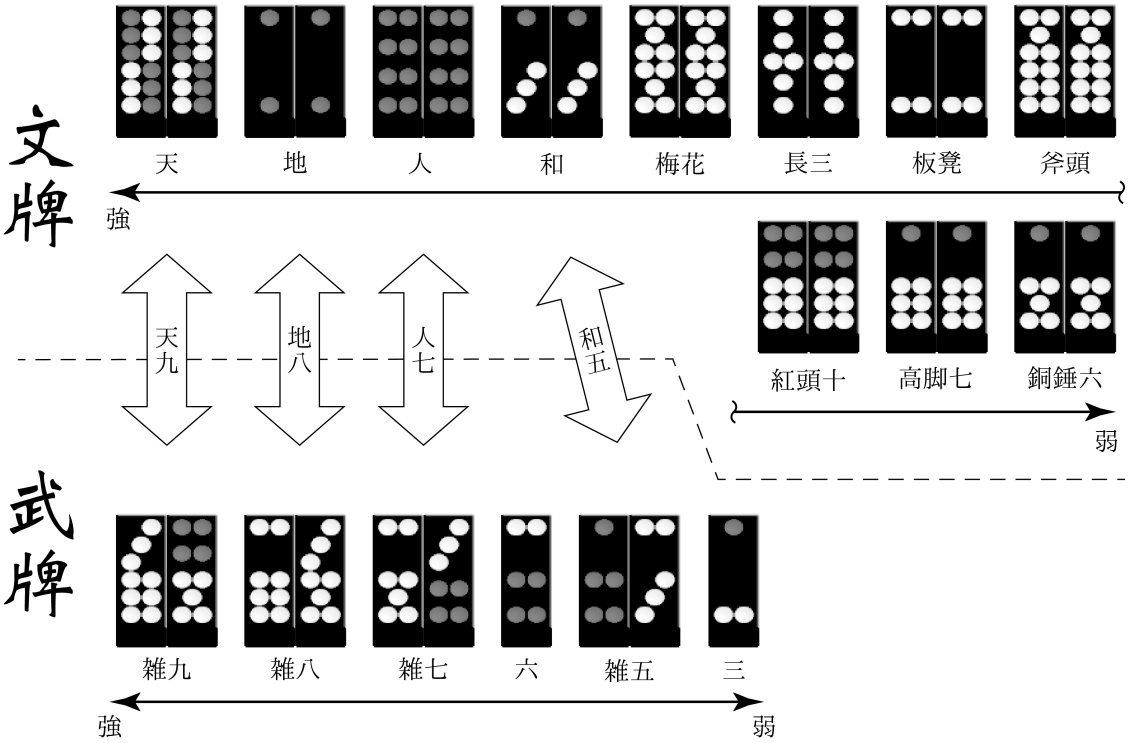


図 2　天九牌を構成する 32 枚

ちなみに，牌の目とスートやランクには特に法則はないので覚えるしかありません．大体表を見ながらやっていると何局かやったところで自然に覚えてきて，やがて牌をパッと見ただけで文牌か武牌か区別できるようになります*64．一応簡単に把握する方法はあり，天・地・人・和は単に覚えるとして，文牌の残りは「まず残っているゾロ目 3 種が大きい順に並び，次いでゾロ目でない 6 がつくものが大きい方から 2 種，最後に 1 のつくものが大きい方から 2 種」という具合に認識しておくことが可能です．

牌のほかには，最初の親や配牌の取り方を決めるのにサイコロを 2 つ，得点を保持しておくのに点棒かポーカーチップを用意するとよいです．

5.2. ゲームの手順

天九は，麻雀の半荘のように東 1–4 局，南 1–4 局の 8 局，または西場や北場も含めた全荘

64 ヒトの学習能力というのは面白いもので，誰でもしばらくやっているとひらがなとカタカナくらいパッと違う系統として文牌と武牌が区別できるようになります．

の 16 局からなります[*65]．各局では 1 人が勝ち，それにより得点がプレイヤーの間で移動します．親が勝ったら同じ局を繰り返す連荘になるのも麻雀とよく似ています．最後の局が終わった時点で最多得点の人が優勝です．

最初にサイコロを振るなど何らかの方法で親を決めたら東 1 局を始めます．局の開始時に 32 枚全部の天九牌をよく混ぜ，8 枚ずつを各プレイヤーの配牌[*66]にするように分けます．（本当はこの牌の分け方が特徴的なのですが，紹介するには結構長いので省略します．何らかの方法で 8 枚ずつに分ければ差し支えありません）．

記事冒頭で述べたように天九はトリックテイキングゲームの一種で，1 局が複数回の**トリック**から成ります．各トリックでは 1 人最初に手牌から牌を出す（**打ち出し**をする）プレイヤーがいて，そのプレイヤーから順に各プレイヤーが反時計周りに順番に出していきます．出せる組み合わせは後述しますが，打ち出しをする人は 1 枚以上 4 枚以下を手牌から出します．2 番目以降のプレイヤーは，最初に出された牌の枚数と同じ枚数を各々が出します．そして全員が 1 回ずつ出したらその中で強弱をもとに誰が勝ちか（＝トリックを取ったか）判定し，そのトリックを取った人が場に出ている牌を自分の獲得した分として 4 枚を 1 山として積んで手元に置いておきます．トリックを取った人が最初に牌を出す人となって次のトリックに移ります．

そして，これが天九のかなり特徴的なルールなのですが，最後のトリックを取った人がその局で勝ちで，その勝ち負けを元に得点を移動します（計算方法は後述）．ただし，次のような制約があります：

> 各自が手牌 1 枚で最後のトリックを迎えた場合（すなわち最後のトリックが 1 枚ずつの勝負になる場合），その局でまだ 1 度もトリックを取っていない人はその最後のトリックへの参加権がなく，自動で負け扱いになる．

単に最後のトリックを取った人が局の勝ちというルールだと強い牌を配られた人がそれを温存するだけで自動で勝ててしまったりして面白くないのですが，この制約によって温存がうまくいくとは限らず，戦略が非自明になっているわけです．

そして，局の勝敗が決定したら，得点を移動したあと，勝った人が親になって次の局を始め

---

65 麻雀よりも 1 局がずっと短いので，16 局行なうことも一般的です．

66 麻雀を知らない方向け： 各プレイヤーの手元にありそのプレイヤーにしか見えない手札のことを**手牌**といい，最初に配られた直後の手牌のことを**配牌**といいます．

ます*67．親が勝った場合は再度同じ局を繰り返す連荘になります．

5.3. トリックに於ける牌の出し方と勝敗の決め方

トリックに於いてはまず最初のプレイヤーが打ち出しとして 1–4 枚の牌を出すのですが，このときに出せる牌の組み合わせには以下のパターンがあります：

- 1 枚出し：
  - 文牌 1 枚
  - 武牌 1 枚
- 2 枚出し：
  - 銅錘六 2 枚を除く，同一の文牌 2 枚
  - 同一の武牌 2 枚
  - 組み合わせ可能な文牌 1 枚と武牌 1 枚： 天九，地八，人七，和五
  - **文尊**（ウェンズン）： 銅錘六 2 枚
  - **武尊**（ウーズン）： 三と六
- 3 枚出し：
  - 組み合わせ可能な文牌 2 枚と武牌 1 枚： 天天九，地地八，人人七，和和五
  - 組み合わせ可能な文牌 1 枚と武牌 2 枚： 天九九，地八八，人七七，和五五
- 4 枚出し：
  - 組み合わせ可能な文牌 2 枚と武牌 2 枚： 天天九九，地地八八，人人七七，和和五五

わかりやすさのためそれぞれの組み合わせを描いたものを図 3 および図 4 に掲げました．文尊と武尊は合わせて**至尊**（ジーズン）と呼ばれ，少し特殊な組み合わせです．至尊以外のパターンはそのパターン内で強弱が定義されており，図中で左ほど強い組み合わせです．

至尊が打ち出しの場合は少し特殊なので後で触れるとして，ひとまず文尊か武尊以外のパターンを最初のプレイヤーが打ち出したとします．2 番目のプレイヤーは，この打ち出しに続いて牌を出します．打ち出しと同じ枚数の牌を出す必要があり，以下の 2 パターンのいずれかの出し方が可能です：

- 打ち出しと同パターンでかつ真に強い牌の組み合わせを表向きで出す．
- 打ち出しと同枚数の牌をどんな組み合わせでもよいので裏向きで出す．

---

67 麻雀と同様に順番に決まった局で親をやる流儀のルールもあるようですが，個人的には勝った人が次の親になるルールの方が逆転要素が大きく面白いのではないかと思います．

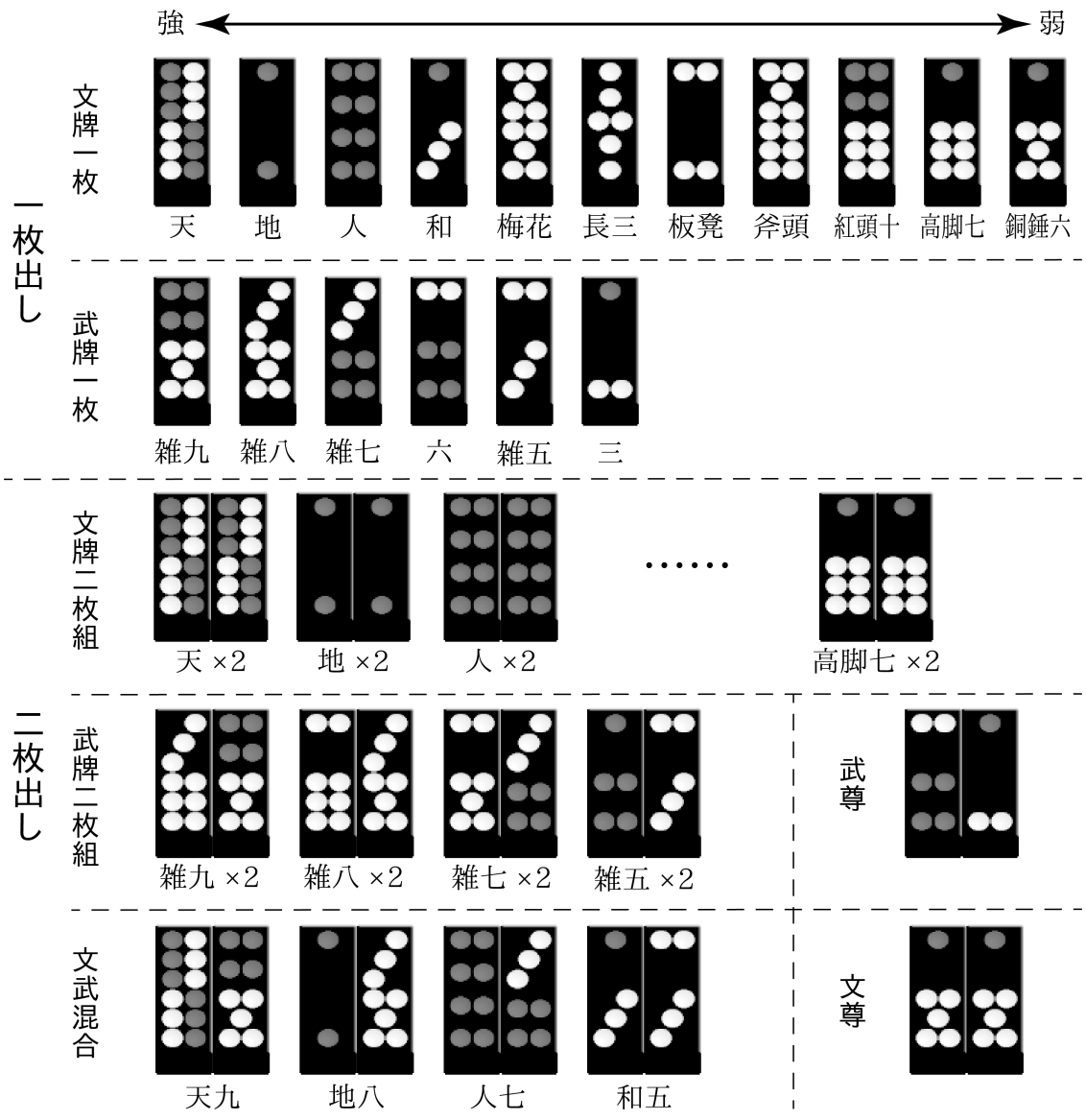


図 3　牌を出すときの組み合わせ（1 枚出し・ 2 枚出し）

裏向きで出された牌はそのトリックで必ず負けることを表します． つまり, 2 番目のプレイヤーは，勝てる組み合わせを出す場合は表向きにして出し，勝てないか或いは勝たなくてよいとして見送る場合はどんな牌か他のプレイヤーに知らせずに裏向きに出してそのトリックの勝負から降りるということをします. 3 番目と 4 番目のプレイヤーも同様に，それより前のプレイヤーの出した組み合わせに勝てる組み合わせを表向きで出すか，或いは勝負から降りて裏向きに同枚数を出します．最も強い組み合わせを出したプレイヤー，すなわち最後に表向きで出

したプレイヤーがそのトリックを取ります.

続いて打ち出しが至尊の場合についてです. これらは或る種の “最強役” で, 武尊（三と六の組み合わせ）が打ち出しの場合, 2 番目以降のプレイヤーは裏向きで 2 枚出す以外のことができません. したがって最初に武尊が打ち出された時点でその打ち出しのプレイヤーがそのトリックを取ることが確定します. 文尊（銅錘六 2 枚）もおおよそ武尊と同様の “最強役” で, 基本的には 2 番目以降のプレイヤーは裏向きで 2 枚出すことになるのですが, 高脚七 2 枚の組み合わせだけには例外的に負けます. つまり, 他のプレイヤーは高脚七 2 枚を持っていたら出して勝つことができます. この高脚七で文尊に勝つことを**擒尊**（チンズン）といいます. 以上が至尊を打ち出したときのルールです. あくまで打ち出しで “最強役” なのであって, 2 番目以降のプレイヤーが至尊の組み合わせを表にして出せることはないので注意してください.

5.4. 得点の移動

得点の移動は, 既に述べたように局が終わってどのプレイヤーが勝ったか決まったときに行なわれるほか, 特定の牌の組み合わせで勝ったときにも発生します. これらについてそれぞれ説明します.

局の終了時の得点移動

局が終わったら, 負けた人は, 基本的には取った山（＝牌 4 枚をひとかたまりにして積んだもの）の数に応じて次のように点数を払います：

- 1 山も取れなかった人は 5 点を場に出す.
- n 山（$1 \le n \le 4$）取って負けた人は, $(4 - n)$ 点を場に出す.
- n 山（$5 \le n \le 6$）取って負けた人は, $(n - 4)$ 点を場から取る.

特に, 4 山取って負けた人は点の増減はなく, 5 山以上取った人は負けでも点が増えます. 1 山も取れなかった人は “追加の罰金 1 点がある” ことに注意してください. 子が全員精算を終えたら, 勝った人は場に出ている点を全部受け取って得点の移動は終了で, 次の局に移ります. ただし, m 連荘（連荘していない親を 1 連荘と扱います）していた親が負けた場合, 親の支払いは $(m + 1)$ 倍になります. また, m 連荘していた親が更に勝った場合, 負けた子の支払いはそれぞれ $(m + 1)$ 倍になります. すなわち, 連荘していない親の関わる得点の移動は 2 倍になり, 2 連荘していた親の関わる得点の移動は 3 倍になり, ... といった具合です. なお, 最初の東 1 局の親は “前回の局で勝って得た親ではない” という気持ちを反映して得点の移動を 2 倍

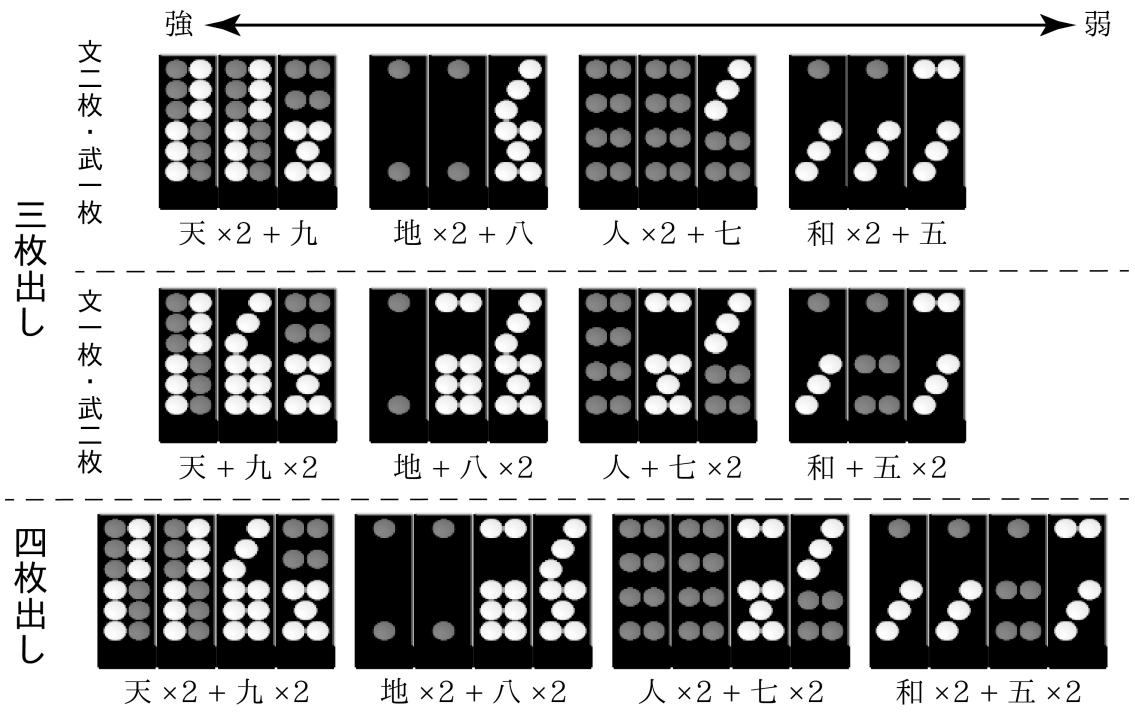


図 4　牌を出すときの組み合わせ（3 枚出し・4 枚出し）

しない，すなわち $m = 0$ として扱います.

最後以外のトリックでの得点移動

局の最後以外のトリックで，以下のいずれかの形でトリックを取った場合，そのトリックを取った人は他の人から 2 点ずつをその場でもらいます[*68]：

- **賀尊**（ハーズン）：　至尊，または高脚七 2 枚で文尊を擒尊したことによる勝ち
- **四大賀**（スーターハー）：　4 枚出しの勝負での勝ち

こちらも局の終了時の得点移動と同様に m 連荘中の親が関わる移動は $(m + 1)$ 倍になります. すなわち，親が上記のどれかでトリックを取ったとき子はそれぞれ $2 \cdot (m+1)$ 点を支払わねばならず，子が上記のどれかでトリックを取ったとき親は $2 \cdot (m+1)$ 点を支払わねばなりません.

最後のトリックによる得点移動の変化

局の最後のトリックの取り方によっては通常の勝敗による得点移動が変わることがあります.

68　俗にこの得点を**ご祝儀**などと呼びます.

具体的には， 最後のトリックが以下のいずれかの場合は得点移動がそれぞれ 2 倍になります（親の $(m+1)$ 倍とも複合し，$2 \cdot (m+1)$ 倍になります）：

- **包尊**（パオズン）： 至尊，または高脚七 2 枚で文尊を擒尊したことによる勝ち
- **四大包**（スーターパオ）： 4 枚出しの勝負での勝ち
- **么結**（ヤオジエ）：
  - 三 1 枚を打ち出してそれで勝った場合
  - 三 1 枚が打ち出され，六 1 枚で勝った場合
  - 銅錘六 1 枚を打ち出してそれで勝った場合
  - 銅錘六 1 枚が打ち出され，高脚七 1 枚で勝った場合

5.5. その他のコーナーケース的ルール

たまにしか起きませんが，コーナーケースとして或る 1 人のプレイヤーが 7 山取って他のプレイヤーが 1 山も取らずに最後のトリックを迎えることがあります．このとき，通常のルールに準拠すると最後のトリックはその 7 山取ったプレイヤーしか参加権がないため自動的に 8 山目も取って勝ちなのですが，この状況だけ例外的に最後のトリックを全員参加にするルールがあります．7 山取った人のその局での勝ちは確定にしつつ，最後の 1 枚も全員参加でトリックを行なってやはり 7 山取った人が勝つかどうかを決定します．ここで 7 山取った人が更に勝ったら**八枝結**（パージージエ）といって全勝で，得点が 4 倍になります．一方最後のトリックだけ取れなかった場合，これも 7 山取った人が勝ちですが，**七枝結**（チージージエ）といって得点が 2 倍になります．

6. ゲームサーバとクライアントの設計

さて，ようやく天九が対局できるゲームサーバを Sesterl（と Elm）で実装する話です．ここに至るまで長すぎじゃない？ もはやこの章がおまけまである．最終的な実装は以下のリポジトリにあってローカルで動かしたり AWS の EC2 インスタンスにデプロイできるようになっているので，もし（万が一）興味があったら覗いてみてください：

```
https://github.com/gfngfn/game_tianjiupai
```

実際に対戦している様子は図 5 のような具合です．

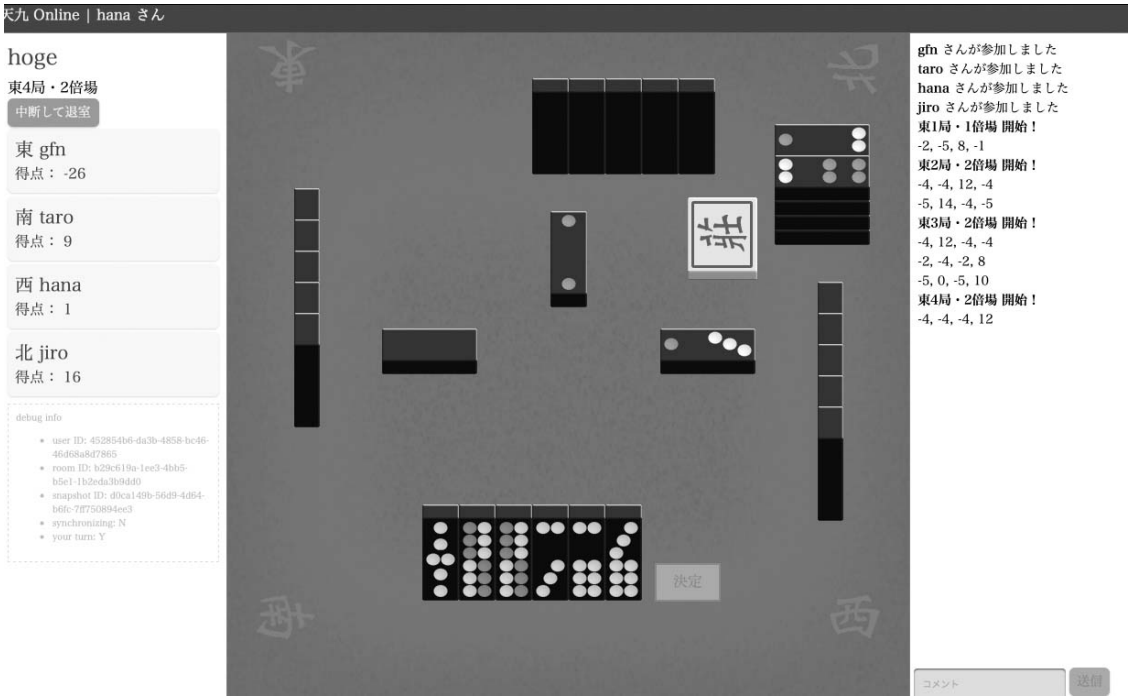


図 5　対局時の様子

6.1. 要件

比較的小規模に作り始めて少しずつ改良していこうということで，ゲームサーバの要件は以下のようにしました：

- 可用性やスケーリングはひとまず追求しないことにし，複数台で分散処理せず 1 台だけで稼働するものとしてつくる．
- ユーザの取り扱いについては，ひとまず永続化されるアカウントの機能などは設けず，ユーザが最初のアクセスでユーザ名を入力したら使い捨てのユーザ ID とセッション変数を発行し，そのセッション変数を cookie に格納させることでユーザとして識別可能にする．
  - 勿論，なりすまし防止のためセッション変数は予測不可能なものを発行する．
  - ユーザは cookie を空にしたり UI 上からログアウトを指示することでユーザとして識別されていない状態に戻ることができる．
  - 1 日などしばらくアクセスのなかったユーザは識別に必要な情報を消してよい．
- 天九が対戦できる場所として**部屋**の概念を設ける．

    - ユーザ ID が発行されたユーザに対しては，最初に既にある部屋の一覧が UI 上に表示されるようにする．
    - ユーザは誰でも新しい部屋を作成したり，空席のある既存の部屋に入ることができるようにする．
    - 1 つの部屋に 4 人集まったらゲームが開始し，すぐに配牌が配られるようにする．
    - ユーザはゲーム途中でも部屋から退出できるようにする．3 人以下になった部屋は新たに人が入ってきて 4 人になるまでゲームが中断されるものとする．
    - ゲームの途中で特定のユーザの接続切れがあったら，復帰するまでしばらく待つ．しばらく待って復帰しなかったら部屋から退出したものと扱う．
- サーバからユーザ側への情報の通知（他のプレイヤーが牌を出したなど）は，WebSocket によってサーバ側から能動的に行なう．
    - HTTP のみによるポーリングでは通信の発生する頻度が高すぎるか或いはリアルタイム性が低くなってしまうため．

また，ユーザは次のいずれかの状態の間を遷移し，それが UI 上で見えるものとします：

(1) まだユーザとして識別されていない状態．トップページが見えており，フォームにユーザの表示名を入力して「開始」のボタンを押すと (2) に遷移する．

(2) ユーザとして識別されているが，どの部屋にも入っていない状態．今どんな部屋があって各部屋にどんなユーザがいるかの一覧が見えており，一覧は接続中リアルタイムで更新される．一覧から空席のある部屋を選ぶとその部屋に入って (3) に遷移するほか，既存の部屋の一覧だけでなく部屋を作成するフォームもあり，新しい部屋を追加できる．また，「ログアウト」を押すとユーザとして識別されなくなり (1) に遷移する．

(3) 特定の部屋に入っている状態．テーブルが見えている．対戦中であればリアルタイムでテーブルの様子が更新され，自分の手番のときは手牌からいくつかの牌を選んで出すことができる．ゲームが南 4 局まで終わると部屋が消え，対局を終えたユーザは (2) に遷移する．また，開始前・開始後にかかわらず，「退室」を押すと (2) に遷移する．

6.2. 設計

さて，要件に基づいてゲームサーバを設計してみましょう．OTP Design Principles の監視ツリーに基づく Erlang 流の設計を行ない，その設計にしたがって Sesterl で実装できること，そ

の際に型をつけるのが難しい部分はフォールバックして Erlang で直接書くなどして Sesterl で書いた部分と共存できることを確かめます*69．以下のように考えて設計します（この設計手順は全部 Erlang で直接書く場合も全く同様です）：

- ユーザ情報は今のところ RDB などに入れて永続化する必要がないので，1 ユーザに対してそのユーザを識別している間だけ存在するプロセスを設けてそのユーザに関する情報をオンメモリで保持させ，ユーザが 1 日などしばらくアクセスして来ずにタイムアウトした場合は消えるようにする．このプロセスの動作を実装したモジュールを `UserServer` と名づけ，`gen_server` コールバックモジュールとして実装することにする．
- `UserServer` を動的に生成・終了するために，それらのスーパーバイザを実装したモジュールである `UserServerSup` を設ける．
- 部屋情報も今のところ永続化する必要がないので，やはり部屋が存在している間だけ稼働して各プレイヤーが牌を出す処理を受けつけそれに応じて全プレイヤーに更新された新しいテーブルの状況を返すなどの処理を行なうプロセスを設け，`RoomServer` というモジュールで `gen_server` コールバックモジュールとして実装することにする．
- `RoomServer` も動的に生成・終了されるので，それらのスーパーバイザを実装したモジュール `RoomServerSup` を設ける．
- 現在存在している部屋一覧を管理するためのプロセスをシステム中にただ 1 つ設け，`PlazaServer` というモジュールで `gen_server` コールバックモジュールとして実装することにする．
- 1 件 HTTP リクエストがきたときに 1 つ立ち上がり，リクエストの内容に応じて動作し，レスポンスを返すと消えるという挙動をするプロセスを設け，`tianjiupai_rest` と名づけたモジュールで取り扱う．
  - これは **Cowboy** [11] という Erlang のライブラリを使って実現し，その都合上型がつけにくいのでひとまず Erlang で直接書く．
- 各々のクライアントと WebSocket の接続が確立されたときに 1 つ立ち上がって，その接続を介した双方向の送受信を担い，接続が切れると消えるという挙動をするプロセスを設け，これを `tianjiupai_websocket` と名づけたモジュールで取り扱う．
  - これも Cowboy を用いて実装しやすくするため直接 Erlang で書く．

69 勿論のことながら，実際の流れとしてはこれができるように適宜 Sesterl の言語機能を拡張していったという方が実態に近いです．

- サーバからクライアントへの能動的な通知のうち，対局中の状態の更新など特に同期していることが重要な内容については，クライアントからサーバへ ACK に相当する返信を WebSocket 接続を通じて行なう．

このように考えてできたのが図 6 の模式図に表したプロセス構造です．角丸の四角形がプロセスの種類を表し，黒の矢印が監視ツリー上の親子関係，灰色の実線矢印がメッセージの送受信の関係を，灰色の点線矢印がプロセス間のモニタ関係を表します．モニタ関係は矢印の起点がモニタする側（`monitor/2` を呼び出す側），先端がモニタされる側であり，`{'DOWN', …}` のメッセージが送られる向きは矢印とは逆向きです．

6.3. 各プロセスの役割

簡単のため，モジュール X に基づいて動作するプロセスを単に X とも呼ぶことにします．

HTTP Handler (`tianjiupai_rest`)

- 個数や生成 / 終了のタイミング： 処理している HTTP リクエストの分だけ存在する．HTTP リクエストが送られてきたときに立ち上がり，対応するレスポンスを返して消える[*70]．
- 主な役割： HTTP リクエストの内容に対応する動作を他のプロセスに依頼し，結果を受け取ってレスポンスを返し，終了する．
- 主な動作：
  - ユーザ作成の POST リクエストが来たとき： リクエストヘッダの `Cookie` が空または無効なセッション変数の場合，ユーザ ID を新規に発行し，対応する `UserServer` プロセスの作成を `UserServerSup` に依頼する．そしてセッション変数を発行する別の機構[*71]に発行を依頼し，これをレスポンスヘッダの `Set-Cookie` に格納してクライアントに覚えさせる．一方で有効なセッション変数がリクエストヘッダの `Cookie` に格納されていた場合は，既に作成済みのユーザ情報を返す．
  - 部屋作成の POST リクエストが来たとき： 部屋 ID を発行し，対応する `RoomServer` プロセスの作成を `RoomServerSup` に依頼する．
  - 部屋一覧を取得する GET リクエストが来たとき： `PlazaServer` に現在の部屋一覧を

70 図 6 には描いていませんが，このような処理は `Cowboy` が行ないます．

71 これは `cowboy_session` [1] というモジュールに依存しています．

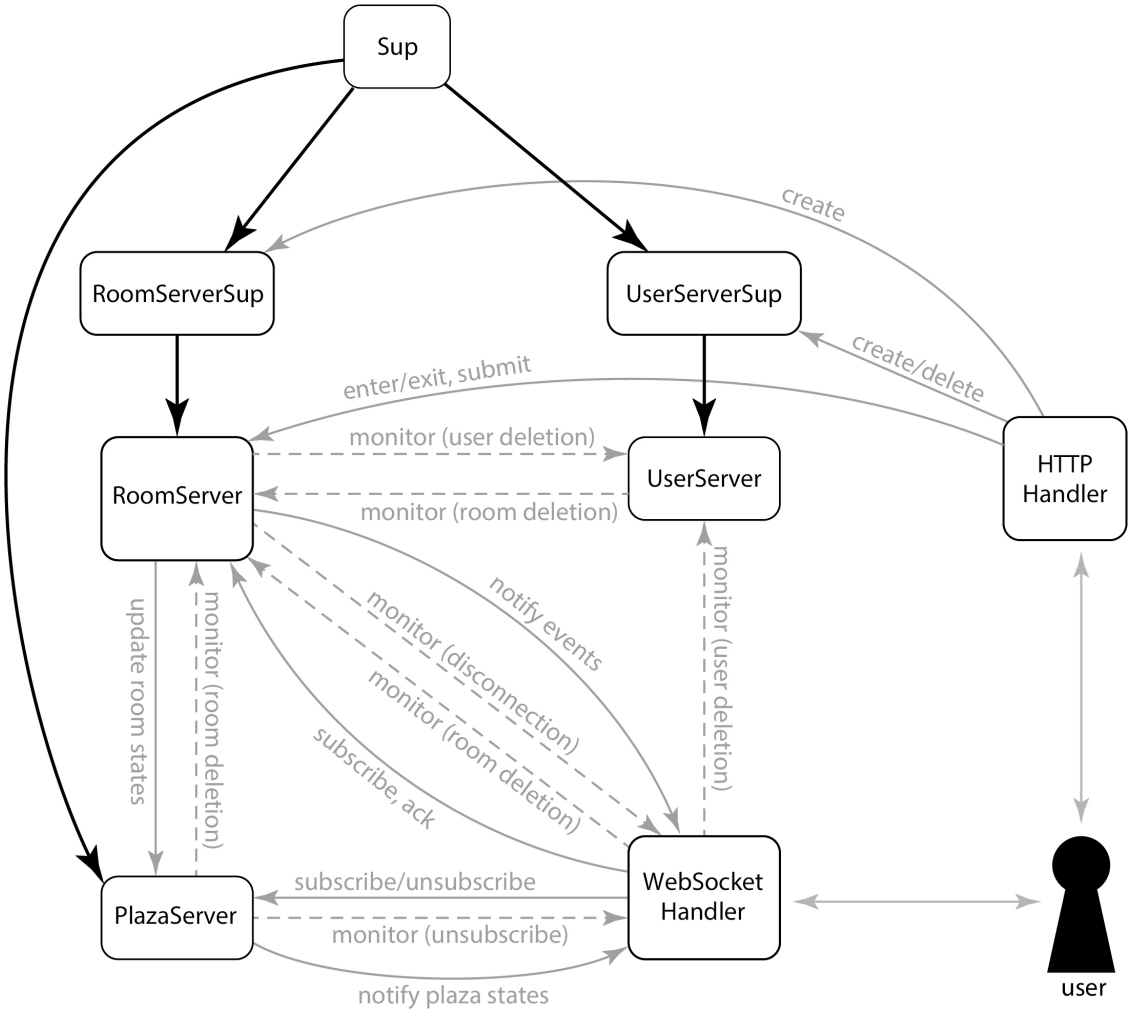


図 6　ゲームサーバのプロセス構造

問い合わせ，結果をレスポンスにして返す.

- 特定の部屋に入る / 部屋から出る旨の PUT リクエストが来たとき：　`RoomServer` にその旨を送り，結果をレスポンスにして返す．また，ユーザに対応する WebSocket Handler の PID を取得し，`PlazaServer` に unsubscribe/subscribe の指示を出す.
- プレイヤーから場に牌を出す POST リクエストが来たとき：　`RoomServer` にその旨を送り，結果をレスポンスにして返す.

WebSocket Handler (`tianjiupai_websocket`)

- 個数や生成 / 終了のタイミング： WebSocket で接続しているユーザの分だけ存在する．WebSocket の接続確立時に立ち上がり，接続が切れると消える．クライアント側から接続を切った場合はその時点でプロセスが終了し，サーバ側から（何らかの不都合などにより）能動的に終了してもやはり接続が切れる[*72]．
- 主な役割： ユーザとの WebSocket での接続を管理し，クライアント側から受信した内容に応じて適切にその旨を他のプロセスに依頼し，また他のプロセスから依頼された内容をクライアント側に送信する．
- 主な動作：
  - 新規の接続があったとき： `PlazaServer` に subscribe の指示を出し，部屋一覧の更新通知を受け取れるようにする．以後，`PlazaServer` から更新通知が来たらクライアント側にその更新内容を送信し，クライアント側では UI 上に表示されている部屋一覧が更新される．
  - クライアント側から「対局中の状態を更新しました」という ACK が送られてきたとき： `RoomServer` にその旨を通知する．
  - 対応する `UserServer` から `{'DOWN', …}` を受け取ったとき： 対応していたユーザがもはやユーザとして識別されなくなったので，自身を終了させて接続を切る．
  - ユーザの属していた部屋の `RoomServer` から `{'DOWN', …}` を受け取ったとき： 部屋が消えたので，ユーザが再度部屋一覧の更新通知を受け取れる状態にするために，`PlazaServer` に subscriber として自身を登録する．

Sup

- 個数や生成 / 終了のタイミング： 全体で 1 つ．常に存在する．
- 主な役割： ゲームサーバをなすアプリケーション全体の監視ツリーの根にあたるスーパーバイザ．`UserServerSup`, `RoomServerSup`, `PlazaServer` を個別に監視する（万が一これらのプロセスが終了したら異常事態なので，その際には再起動する）．

---

72 やはり図 6 には描いていませんが，このような処理は Cowboy が行ないます．

UserServerSup

- 個数や生成 / 終了のタイミング： 全体で 1 つ．常に存在する．
- 主な役割： 必要に応じて UserServer に基づく子プロセスを動的に生成し，それらの子プロセスを監視するスーパーバイザ．子プロセスが正常終了するときはそれに対応するユーザが消えるときなので，再起動しない．

UserServer

- 個数や生成 / 終了のタイミング： 各時点で，ユーザ ID で識別しているユーザ 1 人に対して 1 つ存在する．ユーザが作成されたときに生成され，このプロセスが終了するとユーザが消えたものとする．
- 主な役割： 識別しているユーザ 1 人に対応して存在し，表示名やどの部屋に属しているかなどそのユーザ 1 人に紐づく情報を保持する．
- 主な動作：
  - 対応するユーザがいる部屋の RoomServer プロセスから {'DOWN', …} を受け取ったとき： ユーザがどの部屋にいるかのデータを空にする．

RoomServerSup

- 個数や生成 / 終了のタイミング： 全体で 1 つ．常に存在する．
- 主な役割： 必要に応じて RoomServer に基づく子プロセスを動的に生成し，それらの子プロセスを監視するスーパーバイザ．子プロセスが正常終了するときはそれに対応する部屋が消えるときなので，再起動しない．

RoomServer

- 個数や生成 / 終了のタイミング： 各時点で，存在する部屋 1 つに対して 1 つ存在する．部屋が作成されたときに生成され，このプロセスが終了すると部屋が消えたものとする．
- 主な役割： 部屋 1 つに対応して存在し，どんなプレイヤーがメンバーで，対局開始前なのか，或いは対局が開始しておりどんな牌が場に出ていて各プレイヤーはどんな手牌を持っているのか，などといった状態を保持する．各プレイヤーから送られてきた牌を出すメッセージを処理し，適切に対局状態の更新を行ない，その結果を各プレイヤーに通知する．

- 主な動作：
  - HTTP Handler からユーザが入室するというメッセージを受信したとき： 現在の部屋のメンバーが 3 人以下ならメンバーに加える処理をし，更新された部屋の状態（誰がいるかなど）を `PlazaServer` に伝え，各プレイヤーに対応する WebSocket Handler を介して新規メンバー追加のメッセージを送る．対局開始前でかつメンバーが 4 人になった場合は，対局開始処理をし，WebSocket Handler を介してその 4 人のメンバーに配牌を送る．
  - `tianjiupai_rest` (HTTP Handler) からプレイヤーが牌を場に出すメッセージを受信したとき： そのプレイヤーが手番のプレイヤーであり，かつその牌が手牌に含まれていて出せる組み合わせとしても正しい場合，それを処理して部屋の状態を更新し，新しい状態を各プレイヤーに WebSocket Handler を介して通知する．このとき，各プレイヤーには当然ながらそのプレイヤーから見える情報のみを伝えねばならない（つまり他のプレイヤーの手牌などの情報はクライアントには送らない）．
  - メンバーに対応する WebSocket Handler から `{'DOWN', …}` を受け取ったとき： そのメンバーを接続切れとして扱い，30 秒のタイマーを起動する． このタイマーが切れたら，対応するメンバーは退室したものと扱う．タイマーが切れるまでに再度接続があったら復帰する．

`PlazaServer`

- 個数： 全体で 1 つ．
- 主な役割： どの部屋にも属していないユーザに見せる部屋の一覧のための情報を保持し，各 `RoomServer` から更新情報を受け取り， その更新結果をまだどの部屋にもいないユーザ（に対応する WebSocket Handler）へと通知する．いわゆる **Pub/Sub メッセージングモデル**で定式化しており，`RoomServer` は publisher（発信者）としてこのプロセスへと更新情報を通知し， また WebSocket Handler は subscriber（購読者） としてこのプロセスへと自身を登録することによって更新通知を受け取れるようになる．
- 主な動作：
  - （ユーザが部屋一覧ページを取得するなどして）クライアント側から HTTP Handler を介して部屋一覧の取得リクエストがあったとき： 保持している部屋一覧を HTTP Han-

dler を介して返す.

- publisher である `RoomServer` から部屋状態の更新通知が来たとき： 対応する部屋の情報を更新し，全 subscriber に更新結果を送る．これが WebSocket Handler を通じて最終的にクライアントに届く．今まで保持していなかった部屋の情報だった場合は，その `RoomServer` をモニタする.
- `RoomServer` から `{'DOWN', …}` を受け取ったとき： （部屋が消えたものとして）部屋一覧からその部屋を取り除き，全 subscriber に更新結果を送る.
- （ユーザがどの部屋にも属さない状態になるなどして）WebSocket Handler から更新通知を受け取りたいという subscribe（購読）の旨のメッセージがあったとき： subscriber にそのプロセスを登録し，かつモニタする.
- （ユーザがどれかの部屋に入るなどして）WebSocket Handler から unsubscribe（購読取りやめ）の旨のメッセージがあったとき： subscriber からそのプロセスを取り除く．モニタもやめる.
- WebSocket Handler から `{'DOWN', …}` を受け取ったとき： （ユーザとの接続が切れたものとして）subscriber からそのプロセスを取り除く.

6.4. 主な動作のシーケンス図

プレイヤーが牌を出すときに各プロセス間で起こる送受信の様子は図 7 のシーケンス図に示す通りです．サーバが各クライアントから状態が同期できたことを表す確認応答 (ACK) を受け取るため, 1 往復半または 2 往復の通信が各クライアントとの間で発生します． また， 牌を出したプレイヤーだけは WebSocket ではなく HTTP 通信のレスポンスで状態の更新を受け取るようになっています[*73].

7. まだ納得していないところ (future work)

というわけで Sesterl も Erlang と同様の OTP Design Principles に自然に則ってオンラインゲームを創れる程度には成熟した言語になってきましたが，まあライブラリが全然足りていないとかはともかく，依然として言語としてもっと良くできるなと思うところはあります．顕著

73 現状でも問題ないとは思いますが, HTTP のレスポンスは異常系の場合にしか見ず， 正常系では全員に対して WebSocket で新しい状態を通知する方式に切り替えた方が自然かもしれません.

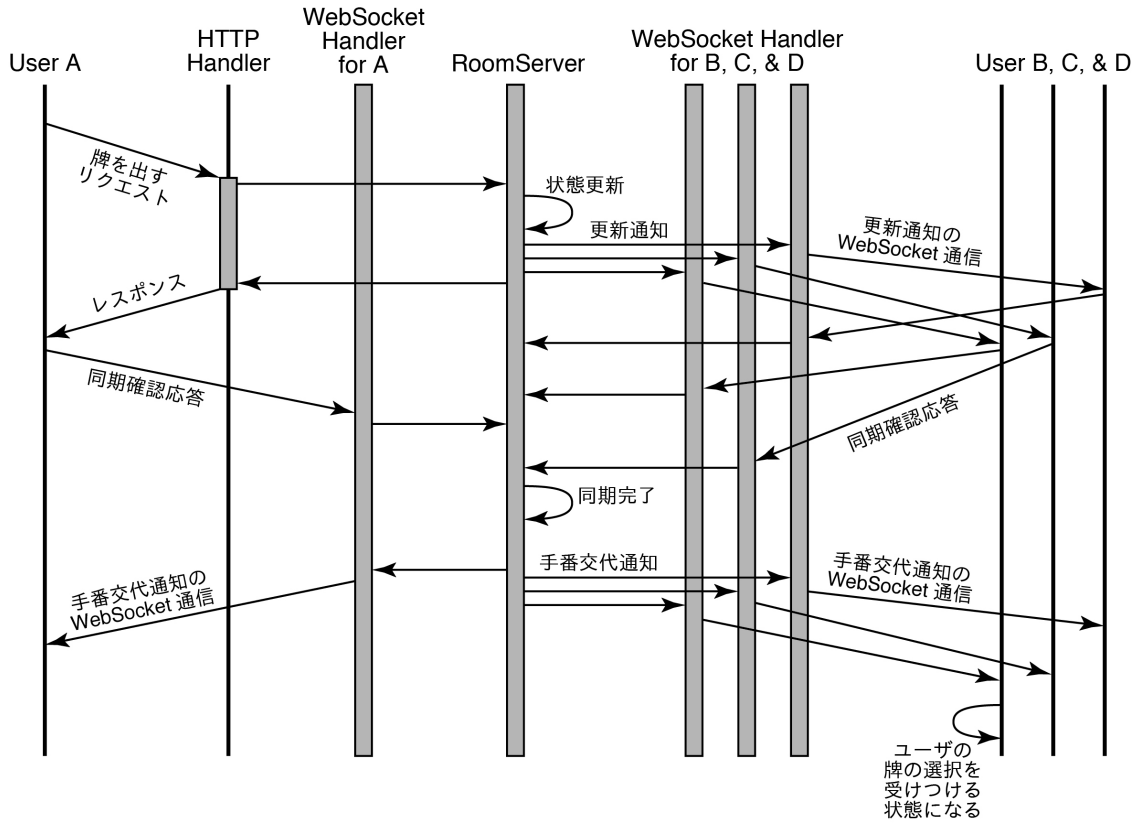


図 7　プレイヤーが牌を出す際のプロセス間のやり取り

なのは以下です：

- 分散処理
    - 複数ノードに分散した処理はまだ想定できていないため，サポートしたいです．現時点でも FFI を使えばいくらでもノード間の RPC などが可能ですが，可能なら言語の水準に組み込みたいと考えています．
- 再帰モジュール
    - これは 4.10 節で述べた通りです．図 6 を見るとわかりますが，天九 Online でもメッセージを相互に能動的に送り合うプロセスを実装したモジュールの組は存在しており，再帰モジュールが使えないがゆえに FFI で型つけを回避して相互依存にしている部分があります．これを FFI に逃げなくてよいように，（前述の通り理論的にかなり大変ですが）再帰モジュールを導入したいと思います．

- GADT による gen_server:call のより強固な型つけ
  - これは GenServer.Behaviour が要請する request 型と response 型の制約が緩いことについてです．4.8 節で紹介したセルプロセスの例はたまたま同期的なメッセージの送受信が Get/Got の組み合わせという 1 種類しかないものでしたが，2 種類以上の同期的なメッセージがある場合は型つけがいい加減になります．人工的な例ですが，以下のように同期的なメッセージが 2 種類ある例を考えてみます：

```
module Callback = struct
  type request =
    | GetNumber
    | GetName

  type response =
    | GotNumber(int)
    | GotName(binary)

  type state = { number : int, name : binary }

  ...
end

module Core = GenServer.Make(Callback)

val get_number(pid) = act
  do res <- Core.call(pid, Callback.GetNumber) in
  case res of
  | Ok(Callback.GotNumber(n)) -> return(Some(n))
  | Error(_)                  -> return(None)
  end
```

プロセスからの返信を受け取る側である get_number の実装では，返ってきたメッセージの形式が必ず GotNumber(n) の形であることを想定していますが， 型検査としては GotName(s) の形である可能性も排除できていません．こうした可能性までしっかり排除できるようにするには GADT (generalized algebraic data type) の機構が必要で，この GADT を用いると以下の擬似コードのような要領で request と response を定義する

ことができ,

```
type number_token =
  | NumberToken /* ダミーの定義 */

type name_token =
  | NameToken /* ダミーの定義 */

type request :: (o) -> o =
  | GetNumber : request<number_token>
  | GetName   : request<name_token>

type response :: (o) -> o =
  | GotNumber(int)  : response<number_token>
  | GotName(binary) : response<name_token>
```

その結果 `handle_call` と `Core.call` の型をより強く制約することができます：

```
val handle_call<$a, $t> :
  fun(request<$t>, pid<$a>, state) -> [info]reply<response<$t>, state>

val call<$a, $t> :
  fun(proc, Callback.request<$t>) ->
    [$a]result<Callback.response<$t>, call_error>
```

これによって `GetNumber` の送信に対しては必ず `GotNumber` が, `GetName` の送信に対しては必ず `GotName` が，それぞれ返信で返ってくることが保証できるようになります.

- session type
  - 既に述べたように，純粋な計算から送受信などの発生する並行処理をモナドによって分離できたのはなかなか有益なのですが, “結局大域的にはどんなプロセスの間でどんな通信が発生しうるのか” の推論やその妥当性の検査はできないので，何らかの方法でそうした推論や検査の機構を備えたいという気持ちがあります．ガチガチに強固に session type の型検査をするのは却って利便性を損なうおそれもあるので，何かしら Gradual Session Types [12] のような機構にすべきかもしれません.

8. まとめ

いかがでしたでしょうか？（言いたいだけ） というわけで本稿ではまず既存言語である Erlang を紹介してそれに型をつけたいという動機を述べ，その目的のために実際に設計・実装した Sesterl という自作言語について各種言語機能を記載し，最後に Sesterl が十分実用可能な水準にあるか確かめるために天九という卓上ゲームのサーバを Sesterl で実装したことを紹介しました．当初 40 ページくらいで書けるかなと思っていたところ 70 ページ強になってしまい随分長くなり書くのも疲れましたが，計算機言語を実用水準まで造り込む初期段階の記録というなかなか見ない文書になったのではないかと思います．

正直なところ全部読み込んで検討してくださった読者はまずいらっしゃらないんじゃないかと思いますが，以下のことだけ伝わっていれば幸いです：

- Erlang の並行・分散処理に特化した言語設計．
- OTP Design Principles という，「ハードウェア障害だけでなく人間が作り込んでしまったバグさえ含め，そもそもプログラムとは変な挙動をすることがつきものである」ということを前提にした可用性上の設計思想．
- Erlang の言語設計や OTP Design Principles の設計思想を活かし，Erlang との FFI での共存も志向しつつ，「とはいえ静的に防げる範囲では型をつけたいよね」という動機によって開発している Sesterl という計算機言語．
- 天九という面白いトリックテイキングゲームがあること．
- 天九のゲームサーバを Sesterl と Elm で実装したこと．

とはいえまあ自己満足という面では存分に充足できたので筆者としては本望です．もし読んで下さった方がおられましたら，最後までおつき合い頂きありがとうございました！

参考文献

[1] Roman Chvanikov (et al.). `cowboy_session`. `https://github.com/chvanikoff/cowboy_session`, 2013.

[2] Evan Czaplicki. `Process – core` *1.0.5*. `https://package.elm-lang.org/packages/elm/core/1.0.5/Process`, 2020.

[3] Derek Dreyer. A type system for recursive modules. In *Proceedings of the 12th ACM SIGPLAN*

International Conference on Functional Programming, pages 289–302, 2007.

[4] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: functional GPU programming in the large. In *Proceedings of the ACM on Programming Languages 2, ICFP, Article 97*, pages 1–30, 2018.

[5] Ericsson AB. `supervisor`. `https://www.erlang.org/doc/man/supervisor.html`, 2021.

[6] Simon Fowler. Typed Concurrent Functional Programming with Channels, Actors, and Sessions. *PhD thesis, University of Edinburgh*, pages 1–280, 2019.

[7] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. *Technical Report NOTTCS-TR-96-3*, pages 1–12, 1996.

[8] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. *IJCAI*, pages 235–245, 1973.

[9] Kohei Honda. Types for dyadic interaction. *Lecture Notes in Computer Science (CONCUR'93)*, 715 , pages 509–523, 1993.

[10] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, 2008.

[11] Loïc Hoguin (et al.). *Cowboy*. `https://github.com/ninenines/cowboy`, 2011.

[12] Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proceedings of the ACM on Programming Languages*, 1 (ICFP) , pages 1–28, 2017.

[13] Hyeonseung Im, Keiko Nakata, Jacques Garrigue, and Sungwoo Park. A syntactic type system for recursive modules. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 46 (10) of *OOPSLA'11*, pages 993–1012, 2011.

[14] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of*

Declarative Programming, pages 167–178, 2006.

[15] Leandro Ostera (et al.). *Caramel*. `https://github.com/AbstractMachinesLab/caramel`, 2020.

[16] Feng Lee (et al.). *Hamler*. `https://github.com/hamler-lang/hamler`, 2019.

[17] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17 (6) , pages 844–895, 1995.

[18] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 568–581, 2016.

[19] Jeremy Pierre (et al.). *Alpaca*. `https://github.com/alpaca-lang/alpaca`, 2016.

[20] Louis Pilfold (et al.). *Gleam*. `https://github.com/gleam-lang/gleam`, 2018.

[21] Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems*, 35 (1) , pages 1–84, 2013.

[22] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, 24 (5) , pages 529–607, 2014.

[23] Takashi Suwa (et al.). *Sesterl*. `https://github.com/gfngfn/Sesterl`, 2020.

[24] Takashi Suwa. *A Tian Jiu Pai Game Server*. `https://github.com/gfngfn/game_tianjiupai`, 2021.

[25] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In *Erlang '10*, pages 23–31, 2010.

[26] 伊藤 拓馬 . アジアゲーム読本 - 第 *1* 集 中国骨牌・天九牌 / 韓国花札・花闘 -. グランペールライブラリー , 2008.

YABAITECH.TOKYO vol.7

2021 年 12 月 31 日 コミックマーケット 99 版 発行

発行者　yabaitech.tokyo
Web サイト　https://yabaitech.tokyo
連絡先　admin@yabaitech.tokyo