OTP in Sesterl:
ML Functors for Typing Erlang/OTP

Takashi Suwa*† (諏訪 敬之)

gfngfn

@bd_gfngfn

*: Kyoto University　†: Imiron

OCaml Meeting 2026　2026-08-22 @ DeNA, Shibuya

誰？

Takashi Suwa （諏訪 敬之）

gfngfn

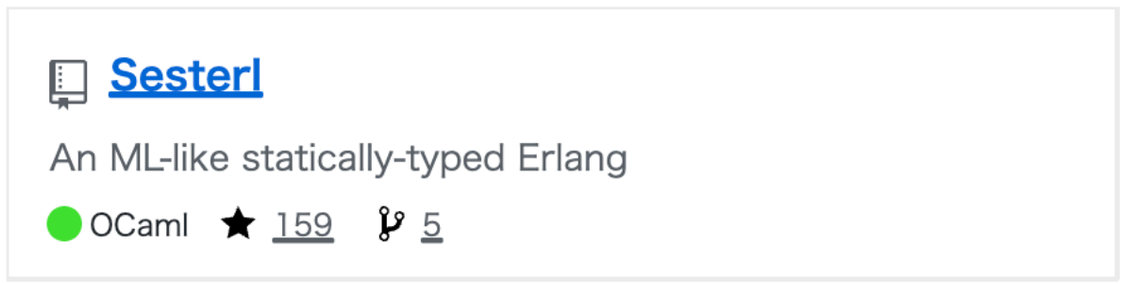
@bd_gfngfn

- 主な制作物：
  - **SATySFi**（静的型つき函数型組版処理システム，2017 年度未踏事業）
- 現所属：
  - 京都大学大学院 五十嵐・末永研究室
    - 自分はメタプログラミング関連の型システムの研究をやっています
  - Imiron Co., Ltd.
    - いわゆる形式手法の社会実装を行なう大学発スタートアップ
      - 時相論理に基づく Cyber Physical System の仕様検証ツール
      - 自動運転車の安全保証
- ここ 2 年ほどは Haskell ばかり書いていて OCaml 書いてない！😡

概要 (1/2)

2020–2023 年頃に，Erlang をラップした ML 風の型つき言語を趣味で設計・実装していました（OCaml 製！）：

https://github.com/gfngfn/Sesterl



主な特徴：

- モナドにより純粋計算と並行処理とを区別 [Orchard & Yoshida 2016] [Fowler 2019]
- わかりやすい FFI により，型なしの通常の Erlang コードと共存可
- F-ing modules [Rossberg, Russo & Dreyer 2014] に基づくモジュールシステム

概要 (2/2)

- Erlang では並行処理を直接プリミティヴで書くことは少なめで,
  OTP というライブラリを用いて書くのが普通
  - 多くの場合，`gen_server` と `supervisor` という 2 つのモジュールを使う
- Sesterl では，この `gen_server` と `supervisor` が
  ML ファンクタの形へと抽象化されている
- さらに，Sesterl プログラムを Erlang でのモジュール単位へと
  コンパイルするために，**ファンクタの適用を型検査時に展開**
  - Erlang のモジュールは入れ子やファンクタなどがない
  - **静的解釈 (static interpretation)** [Elsman, Henriksen, Annenkov & Oancea 2018]
    という手法を用いて実現
- 今回はこれらの定式化の紹介です

▶ 問題意識

- 基本的な言語設計
- 純粋計算/並行処理の区別
- モジュールシステム
- 実際の使用例
- Future Work
- まとめ

Erlang ってどんな言語？ (1/2)

- いわゆる函数型で，破壊的代入は原則できない
- **アクターモデル**に基づく並行並列処理が言語本体に備わっている
- **プロセス**（＝“メモリ共有のない軽量スレッド”）の機構をもつ
  - プロセスはプログラム実行中に動的に生成・終了する
- プロセス間は**非同期メッセージパッシング**によりデータを送受信
  - 送信先の指定などには **PID** (process ID) を用いる

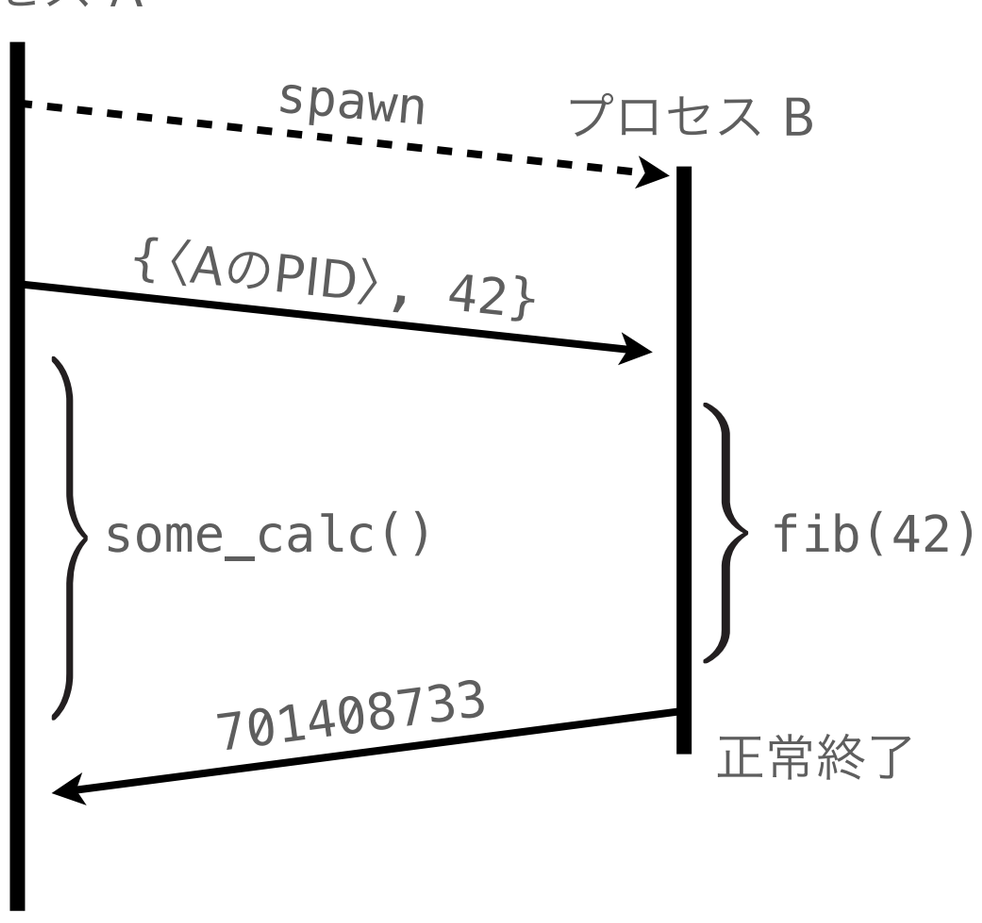
特に並行並列処理のない簡単な例：

```
fib(N) ->
  case N < 2 of
    true  -> 1;
    false -> fib(N - 1) + fib(N - 2)
  end.
```

Erlang ってどんな言語？ (2/2)

```
main() ->
  PidB =
    spawn(fun() ->
      receive
        {From, N} ->
          Result = fib(N),
          From ! Result
      end
    end),

  PidA = self(),
  PidB ! {PidA, 42},
  ResultA = some_calc(),
  receive
    ResultB ->
      {ResultA, ResultB}
  end.
```

- プロセスを生成しその PID を返す：
  spawn(〈生成されたプロセスで走る処理〉**)**
- 受信： **receive** … **end**
- 送信：〈送信先 PID〉**!**〈メッセージ〉



Erlang の利点・弱点

- **函数型で並行並列機能がプリミティヴとして備わった意味論**
  - 競合状態が生じにくい
- **耐障害性の追求しやすさ** (“Let It Crash!”)
  - link/monitor によりプロセスの生死を監視する機能
  - OTP という成熟したプラットフォームの存在

- **実装の正しさを静的に保証する手段の不足**
  - なにしろ型が事実上ない
    - Success typing [Lindahl & Sagonas 2006] という gradual typing のような緩い型システムに基づく型検査器があるが，unsound で実際かなりゆるい
- **言語機能上の細かい厄介さ**
  - プロセスとプロセス名の紐づけに関しては競合状態が防げていない
  - オプション引数が言語自体にはなくライブラリごとに定式化がバラバラ
  - … etc.

目標とする要件と現状

- **sound な型システム**
  - プロセス間通信にどう型をつけるかが難しい
    - アクターモデルは π 計算などと比べても難易度が高め

  ➡ 現状： **比較的素直なモナドによる型つけ**．sound だが柔軟性はやや不足

    - もっと真面目にやるなら session type など（Future work）

- **わかりやすい FFI**
  - 既存の Erlang 製の資産を活かしやすくするため
  - すでに Erlang で書かれているプログラムからも少しずつ移行できる

  ➡ 現状： 我ながら良い感じ 😎

- **現実的なアプリケーションの構築に使用できること**

  ➡ 現状： ちょっとしたオンラインゲームサーバ（1 万行規模）の実装に使えた！

    - とはいえ，どうしても FFI に頼らざるを得ない部分はチラホラある

関連プロジェクト

- **Elixir**
  - 圧倒的実績．Ruby 風の構文，Clojure 風のマクロ機構
  - **長らく型なしだったが，今年 gradual typing が後づけされたらしい**
- **Alpaca** (https://github.com/alpaca-lang/alpaca)
  - ML 風の構文・意味論
  - 型つき，ラベルなし union type も扱えるらしい（型推論はしない？）
  - 直近 7 年ほど更新がない
- **Gleam** (https://github.com/gleam-lang/gleam)
  - Erlang VM とは別に JavaScript もターゲット言語にしている模様
  - インディーズ言語としてはかなりユーザが増えている様子
    - Sesterl と近い時期に開発が始まり，当初からコミュニティができはじめていた
  - **言語水準ではあまり熱心に型つけしない方針らしい**
    - （最近は追えていないので詳細は自信なし）

- 問題意識

▶ 基本的な言語設計

- 純粋計算/並行処理の区別
- モジュールシステム
- 実際の使用例
- Future Work
- まとめ

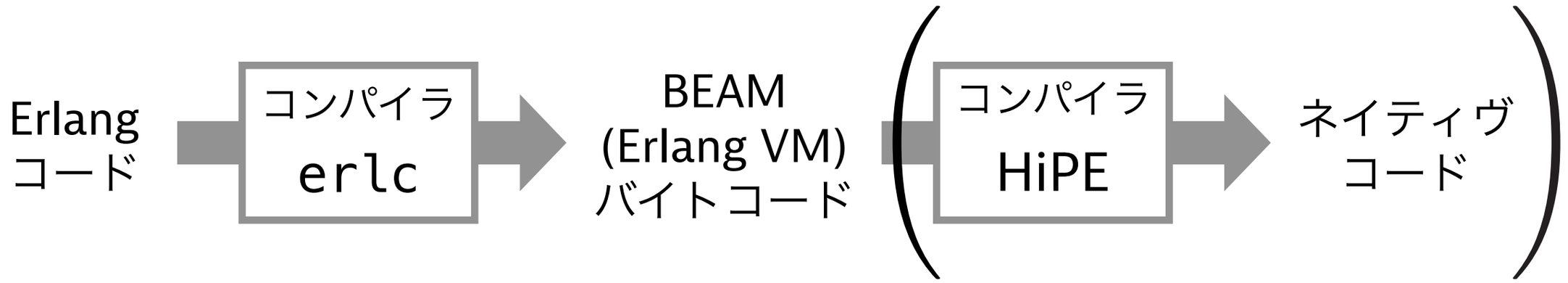
基本的な言語設計

- おおよそ ML 系言語の特徴を踏襲
  - 値呼び
  - Hindley–Milner 多相とその型推論
  - ADT とパターンマッチ
  - ただし具象構文にはあまりこだわらない

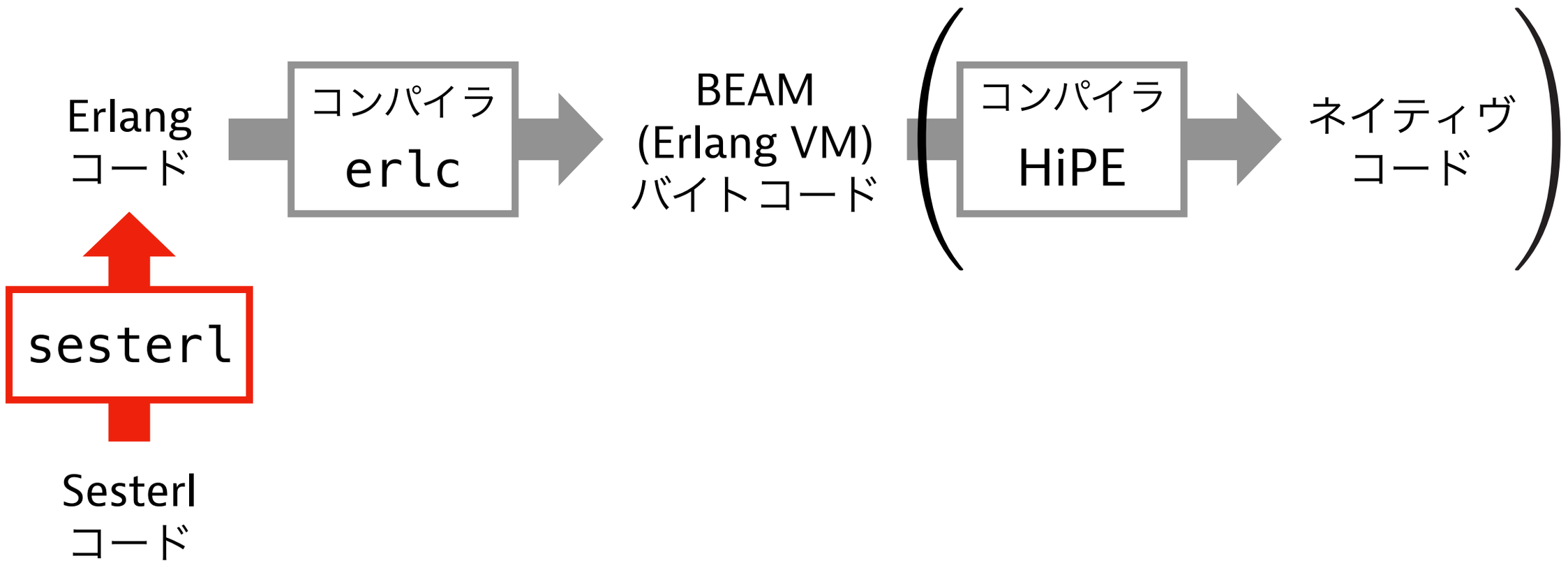
```
type option<$a> =
   | None
   | Some($a)

val get_or_else(v, d) =
  case v of
   | None    -> d
   | Some(x) -> x
  end
```

- Erlang との FFI での連携しやすさを考慮し Curry 化はしない
  - 函数は arity をもつ
  - 型の構文： τ ::= … | **fun**(τ, …, τ) -> τ
  - 例えば **fun**(int, int) -> bool ≠ **fun**(int) -> (**fun**(int) -> bool)

バックエンド


Erlang
コード
コンパイラ
erlc
BEAM
(Erlang VM)
バイトコード
コンパイラ
HiPE
ネイティヴ
コード

バックエンド



ひとまず VM のバイトコードではなく Erlang コードを出力とする

- 函数ごとの FFI が簡潔に実現できる
- 開発の都合： 目視でデバッグしやすい

- 問題意識
- 基本的な言語設計

▶ 純粋計算/並行処理の区別

- モジュールシステム
- 実際の使用例
- Future Work
- まとめ

純粋/並行の区別 (1/3)

- **純粋/並行を型で区別したい ➡ モナドを導入** [Orchard & Yoshida 2016] [Fowler 2019]
- 基本的には $[\tau]\,\tau'$ という型で以下の直観を表現したい：
  - 最終的に τ' 型の値を返すような計算であり，
    メッセージを受信するならそれは τ 型のメッセージでなければならない
- 直接 send/receive を書くことはあまりないが，
  この定式化が結果的に OTP 水準の記述にも有用
- もっと真剣にやるなら session type などを導入すべきだが，
  今のところそこまで必要になっていない

$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash \texttt{return}(e) : [\tau]\,\tau'} \qquad \frac{\Gamma \vdash e_1 : [\tau]\,\tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : [\tau]\,\tau_2}{\Gamma \vdash \texttt{do}\ x\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : [\tau]\,\tau_2}$$

$$\frac{\Gamma, x : \tau \vdash e : [\tau]\,\tau'}{\Gamma \vdash \texttt{receive}\ x\ \texttt{->}\ e\ \texttt{end} : [\tau]\,\tau'}$$ (本当は x はパターンで，branching あり)

純粋/並行の区別 (2/3)

- **ただし，$[\tau]\tau'$ を単独の型とすると今回のケースでは不都合あり**
  - 例： `let comp = (do x <- `e_1` in `e_2`) in` … の e_1 と e_2 は値呼びでも即座に評価されてはならない
  - したがって $[\tau]\tau'$ 型の式のコンパイル結果は（`fun() -> ...`）という thunk にし，bind ごとに逐一 force せねばならない
    - **パフォーマンスが低下**，あるいは少なくとも複雑な最適化を要する
    - **FFI が複雑化**（これは目標の要件に反する）

- **$[\tau]\tau'$ 型は函数の codomain にのみ現れるように制限し，純粋・並行どちらの函数も素直な式にコンパイルする**
  - 型の構文： $\tau ::= \cdots \mid$ `fun(`τ`, ..., `τ`) -> `τ $\mid$ `fun(`τ`, ..., `τ`) -> [`τ`]`τ
  - 式の構文： $e ::= \cdots \mid x \mid$ `fun(`x`, ..., `x`) -> `e $\mid$ `fun(`x`, ..., `x`) -> act `C
  - $C ::= \cdots \mid$ `do `x` <- `C_1` in `C_2 $\mid$ `receive ... end`

純粋/並行の区別 (3/3)

- PID を扱う型にも「そのプロセスに何型のメッセージを送ってよいか」の情報を保持させる
- 型の構文：

$$\tau \ ::= \ \cdots \ \mid \texttt{pid<}\tau\texttt{>}$$

$$\frac{\Gamma \vdash e_0 : \texttt{[}\tau\texttt{]unit}}{\Gamma \vdash \texttt{spawn(}e_0\texttt{)} : \texttt{pid<}\tau\texttt{>}}$$

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

$$\frac{}{\Gamma \vdash \texttt{self()} : \texttt{[}\tau\texttt{]pid<}\tau\texttt{>}}$$

```
val main() = act
  do pidB <-
     spawn(
       receive
         {from, n} ->
           let res = fib(n) in
           send(from, res)
     end)
in
do pidA <- self() in
do send(pidB, {pidA, 42}) in
let resA = some_calc() in
receive
  resB -> {resA, resB}
end
```

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別

▶ モジュールシステム

- 実際の使用例
- Future Work
- まとめ

モジュールシステム

- ご存知の通り，いわゆるカプセル化を型システム的に保証する仕組み
- ここでは **F-ing Modules** [Rossberg, Russo & Dreyer 2014] に基づく定式化を採用
  - ファンクタ，第 1 級モジュールなども自然に扱える
    - 注意： ここでいうファンクタは “モジュールをモジュールに写す大きい函数”
  - 型検査が決定可能
  - 依存型を用いる必要はなく，System Fω のサブセットで事足りる
- **ファンクタにより OTP の `gen_server` や `supervisor` を自然に表現できる**（この後説明）
- **静的解釈 (static interpretation)** [Elsman, Henriksen, Annenkov & Oancea 2018] という手法によりファンクタをコンパイル時に展開する処理も実装
  - Sesterl コードを Erlang のモジュール単位へとコンパイルするのに必要

背景： gen_server とは

- 以下のような機能をもつプロセスを簡単につくるための枠組み：
  - **内部状態**を保持する
  - **call**（**同期通信**）： 他のプロセスからメッセージを受信して，
    必要に応じて状態を更新するなどし，送信元に返信を送る
    - 送信した側のプロセスは，返信が届くまで評価がブロックされる
  - **cast**（**非同期通信**）： 他のプロセスから一方的にメッセージが届き，
    必要に応じて状態を更新したりする
    - 送信した側はブロックされない
  - （他にもいくつか最初から備わっている機能があるが今回は省略）

背景： gen_server の API

- 右の例： 整数 1 個を状態としてもち,
  外部から get/set ができるプロセスの
  gen_server による実装

- **-behavior**(gen_server) は,
  プロセス内で使われる以下のような
  コールバック函数の実装を要請：
  - init/1
    - 起動直後に初期状態を設定
  - handle_call/3
    - call を処理する畳み込み函数
    - 返信内容と更新後の状態を返す
  - handle_cast/2
    - cast を処理する畳み込み函数
    - 更新後の状態を返す
  - （本当は他にさらに 2 つある）

```
-module(cell).
-behavior(gen_server).
-export([
  init/1, handle_call/3, handle_cast/2,
  start_link/1, get_number/1, … ]).

% プロセス内で使われる函数たち：

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

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

handle_cast({set, N}, _State) ->
  {noreply, N}.

% 他のプロセスから呼ぶ函数たち：

start_link(N) ->
  gen_server:start_link(cell, N, []).

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

set_number(Pid, N) ->
  gen_server:cast(pid, {set, N}).
```

gen_server に型をつけてラップする

ML ユーザ的な視点で見ると……

- **コールバック函数の実装を要請する**
  -behavior(gen_server) は
  シグネチャのようなもの
- gen_server:call(…) などの函数は,
  コールバック函数から生成されたもの
  として扱えそう

gen_server は, ファンクタとして
以下のように形式化できそう！

```
module GenServer.Make :
  functor(Callbacks : Behavior) -> sig
    val start_link : …
    val call : …
    …
  end
```

```
-module(cell).
-behavior(gen_server).
-export([
  init/1, handle_call/3, handle_cast/2,
  start_link/1, get_number/1, … ]).

% プロセス内で使われる函数たち：

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

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


handle_cast({set, N}, _State) ->
  {noreply, N}.

% 他のプロセスから呼ぶ函数たち：

start_link(N) ->
  gen_server:start_link(cell, N, []).

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

set_number(Pid, N) ->
  gen_server:cast(pid, {set, N}).
```

ファンクタ GenServer.Make の使用例

先ほどと同じ実装例 Cell：

整数 1 個を状態としてもち，
外部から get/set できる
プロセスを実装したモジュール

- 実際には，コールバック函数に加えて関連する型を指定したモジュールを受け取るようにシグネチャが要請
  - Map.Make が key の型を必要とするのと同様
- start_link, call, cast などを実装したモジュールを返す

```
module Cell = struct
  module Callbacks = struct
    type state = int
    type init_arg = int
    type call_request = Get
    type call_response = Got(int)
    type cast_msg = Set(int)
    val init(n : init_arg) = Ok(n)
    val handle_call(Get, n : state) =
      GenServer.reply(Got(n), n)
    val handle_cast(Set(m), n : state) =
      GenServer.no_reply(m)
  end
  include GenServer.Make(Callbacks)
…
  val get_number(pid) =
    do response <- call(pid, Get) in
    case response of
    | Got(n) -> return(n)
    end

  val set_number(pid, m) =
    cast(pid, Set(m))
end
```

ファンクタをどうコンパイルするか？

Sesterl のモジュール構造をどう出力結果の Erlang にマップするか？
Erlang のモジュールは入れ子がなくファンクタなども到底ない

- **ファンクタ適用を静的に展開すればよい！**
  - ここでは**静的解釈 (static interpretation)** [Elsman, Henriksen, Annenkov & Oancea 2018] による
    - 型検査中にファンクタの中身を持ち回り，適用で instantiate する
    - MLKit や Futhark で使われており，理論的に面白いが複雑なので詳細は省略
  - その代わり，**第 1 級モジュールの自由度は制約される**
    - これまでのところ，第 1 級モジュールの使用で障壁となった場面はない
- 入れ子 `Foo.Bar.Qux` は単に `Foo_Bar_Qux.erl` という具合に
  各々の部分を単一ファイルに落とし込むだけで OK
  - `Foo_Bar.erl` と `Foo_Bar_Qux.erl` は一般には相互依存しうるが，
    Erlang モジュールは普通に相互依存可能なので無問題

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- モジュールシステム

- **▶ 実際の使用例**
- Future Work
- まとめ

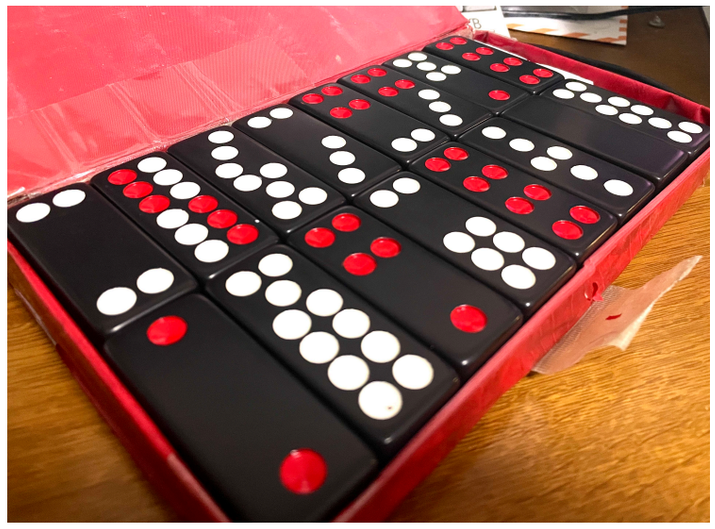
実際の使用例

- ドッグフーディングのために，オンラインゲームサーバを Sesterl で実装してみた

  https://github.com/gfngfn/game_tianjiupai

  - **天九** (Tian Jiu, Tien Gow) オンライン対戦サーバ
    - **天九牌**という 32 枚の牌を使う 4 人対局ゲーム
    - なかなか中毒性があるが，日本では多分数百人の物好きしかやってない
  - HTTP/WebSocket 部分は既存ライブラリ Cowboy を FFI で型つけして使用
  - フロントエンドは Elm 製
  - AWS EC2 にデプロイして動作済
    - 友人らと Discord で通話しながら対局できた



様子

天九 Online | hana さん

a

東1局・1倍場

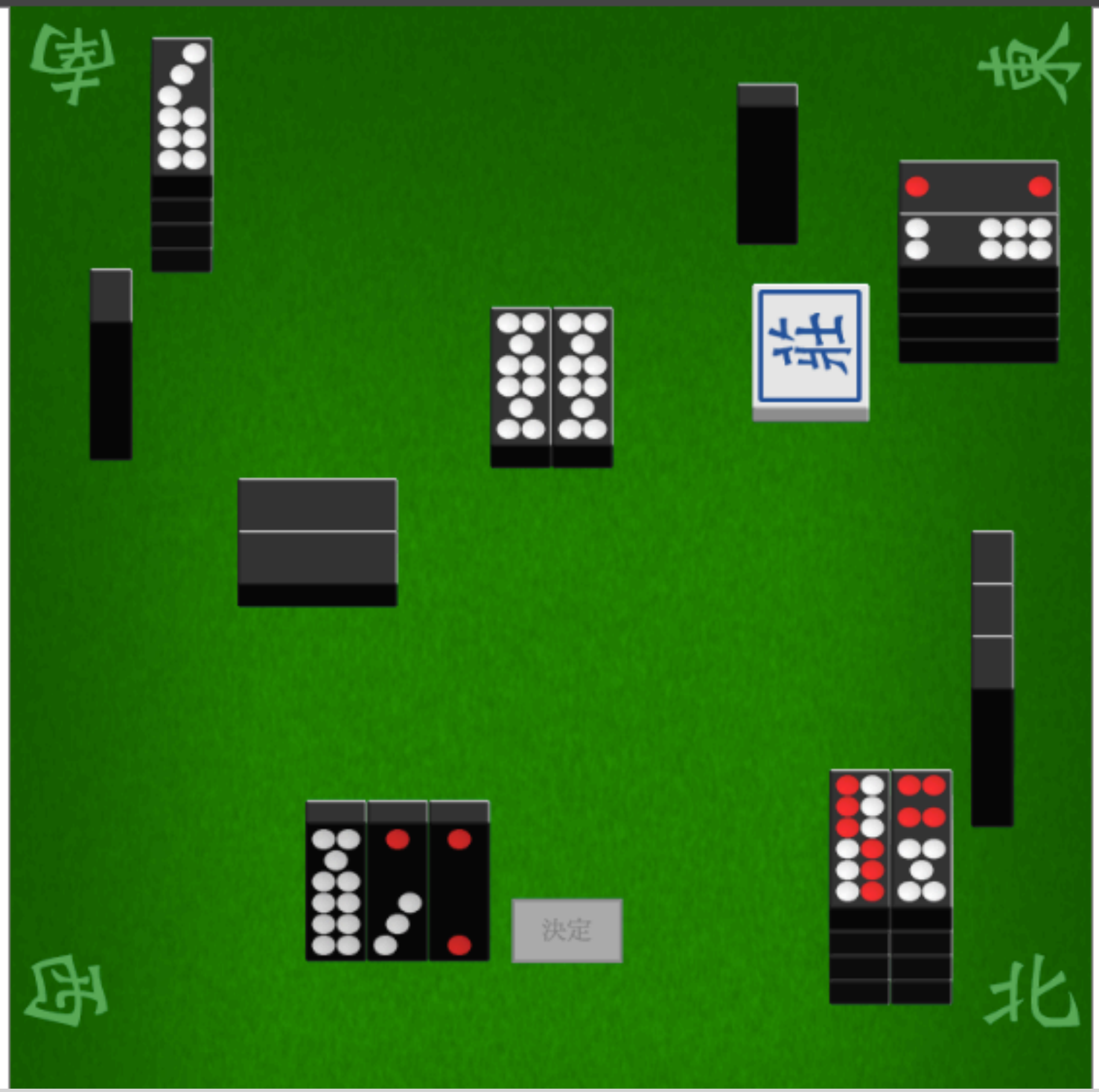
中断して退室

東 taro
得点： 0

南 gfn
得点： 0

西 jiro
得点： 0

北 hana
得点： 0



taro さんが参加しました
gfn さんが参加しました
jiro さんが参加しました
hana さんが参加しました
東1局・1倍場 開始！

debug info

- user ID: 2cae9124-d6aa-4b78-bf4d-1359d8fe1f94
- room ID: da5bf285-b668-42f4-98ac-6c3ddf3399e6
- snapshot ID: 05fa35c3-25c4-4f14-

コードの構成

- いわゆるビジネスロジック自体はほぼ Sesterl で書けた
  - Sesterl コード： 19 ファイル，4291 行
  - Erlang コード： 5 ファイル，648 行
  - 実際にはさらに Sesterl の `stdlib` などに依存
- FFI に頼らざるを得ない部分も若干あった
  - HTTP や WebSocket のルーティング，セッション管理など
  - 相互依存の解消など

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- モジュールシステム
- 実際の使用例
- **▶ Future Work**
- まとめ

Future Work

- **Session type** を入れる
  - メッセージの型だけでなくやり取りの手順まで静的に検査する仕組み
  - 当初はこれが最大の関心だった（Sesterl ← Session-typed Erlang）
  - 最近提案された **Speak Now** [Flowler & Hu 2026] という定式化が有用かも
- または，**GADTs** のサポート
  - リクエスト-レスポンスの対をより厳密に型つけするのに便利
- **再帰モジュール** (**recursive modules**) のサポート
  - 2 種類の `GenServer` 準拠のモジュールに基づくプロセスの間で
    相互に能動的にメッセージを送信したいケースがあり，
    相互依存するモジュールを型検査できるようにしたい
  - 再帰モジュールの静的解釈が地獄
    - おそらくまだ誰も検討していない
    - 一般の再帰＋静的解釈は型検査が決定不能なはずで，何かしらの制限が必要

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- モジュールシステム
- 実際の使用例
- Future Work

▶ まとめ

まとめ

- ML 風の型つき Erlang である **Sesterl** を紹介しました

  https://github.com/gfngfn/Sesterl

- 主な特徴：
  - モナドによる純粋/並行の区別
  - ML ファンクタを備えたモジュールシステムによる **OTP** のラップと，**静的解釈**による Erlang モジュール単位へのコンパイルを実現

- 使用例である Sesterl & Elm 製の天九ゲームサーバ：

  https://github.com/gfngfn/game_tianjiupai

- 今回の内容の詳しい話は昔 Yabaitech Tokyo vol.7 に書きました：

  https://yabaitech.tokyo/assets/pdf/yabaitechtokyo_vol7.pdf

References

- Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. Static interpretation of higher-order modules in Futhark: functional GPU programming in the large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97, 2018.
- Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, University of Edinburgh, 2019.
- Simon Fowler and Raymond Hu. Speak now: safe actor programming with multiparty session types. *Proc. ACM Program. Lang.* **10**, OOPSLA1, Article 159, 2026.
- Benedict Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. *Technical Report NOTTCS-TR-96-3*, Department of Computer Science, University of Nottingham, 1996.
- 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 (POPL '16)*, pp. 568–581, 2016.
- Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of Functional Programming*, **24**(5), pp. 529–607, 2014.

おまけ

ラベルつきオプション引数

```
val is_log_file(path) = String.is_suffix(-target path, -suffix '.log')

/* increment : fun(int, ?diff int) -> int */
val increment(n, ?diff d_opt) =
  case d_opt of
  | None    -> n + 1
  | Some(d) -> n + d
  end

val increment(n, ?diff d = 1) = n + d /* 上と同じ */

val main() = (increment(42), increment(42, ?diff 15)) /* ==> (43, 57) */
```

- 必須： `-foo`，省略可： `?bar`
- 順番は入れ替え可能
- 高階函数にも型がつけられる
  - **列多相** [Gaster & Jones 1996] を応用した独自の型システムによる
  - Curry 化を考慮しなくてよいので，OCaml の場合 [Furuse & Garrigue 1995] より単純