Sesterl:

静的型つきML系AltErlang

2020-10-09

自作プログラミング言語の集い（主催： Opt Technologies）

Takashi Suwa (@bd_gfngfn)

概要

Erlang をラップした ML 風の型つき言語を設計・実装しています

https://github.com/gfngfn/Sesterl

主な特徴：

- モナドによる純粋計算/並行処理の区別 [Orchard & Yoshida 2016] [Fowler 2019]
- ラベルつきオプション引数を多相に扱う 2 階の体系
  - SML# のレコード [Ohori 1995] を独自に応用したもの
- F-ing modules [Rossberg, Russo & Dreyer 2014] に基づくモジュールシステム

▶ 問題意識

- 基本的な言語設計
- 純粋計算/並行処理の区別
- ~~ラベルつきオプション引数~~
- モジュールシステム
- Future Work
- まとめ

Erlang ってどんな言語？ (1/2)

- 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
- いわゆる函数型で，破壊的代入は原則できない

シングルスレッドの簡単な例：

```
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)

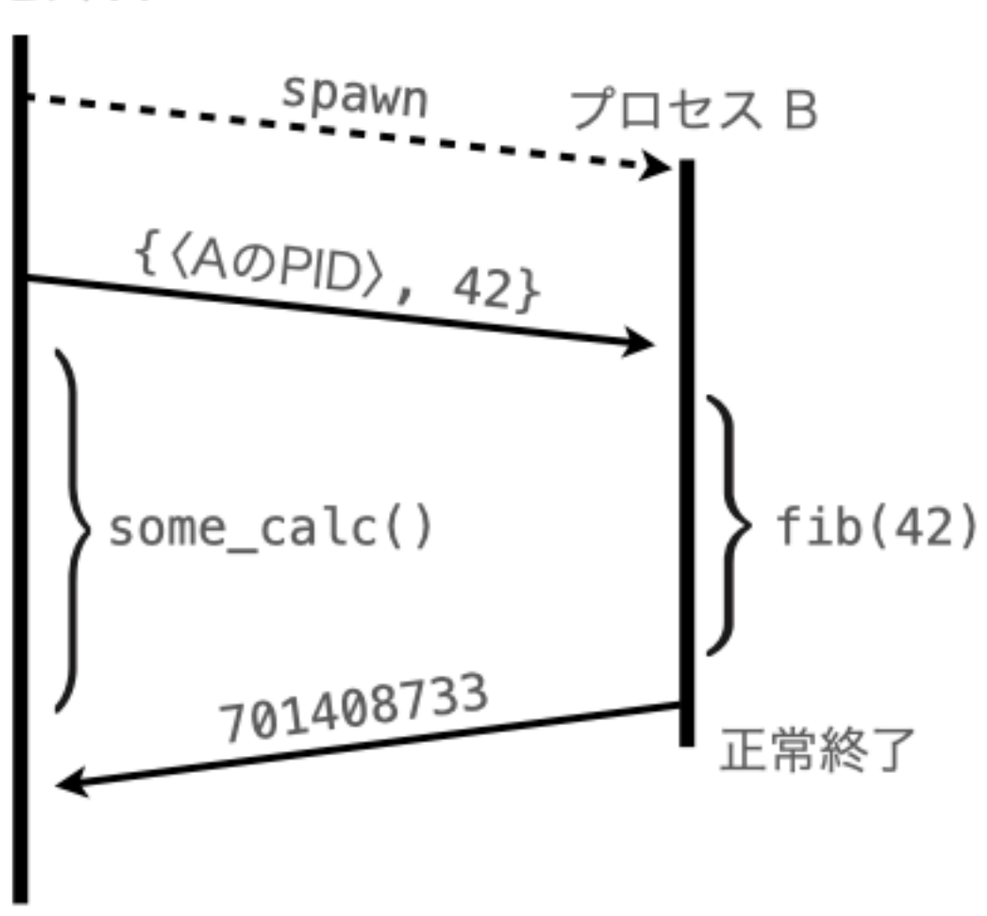
```
main() ->
  PidB =
    spawn(fun() ->
      receive
        {From, N} ->
          Result = fib(N),
          From ! Result
      end
    end),

  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 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（→問題意識）

😊

- **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**
  - 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
- **言語機能上の細かい厄介さ**
  - プロセスの生死に関しては競合状態が防げていない
    - 非同期的にプロセスを終了させ同名で再起動すると確率的にダブるなど
  - 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.

- 問題意識

▶ 基本的な言語設計

- 純粋計算/並行処理の区別
- ~~ラベルつきオプション引数~~
- モジュールシステム
- 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

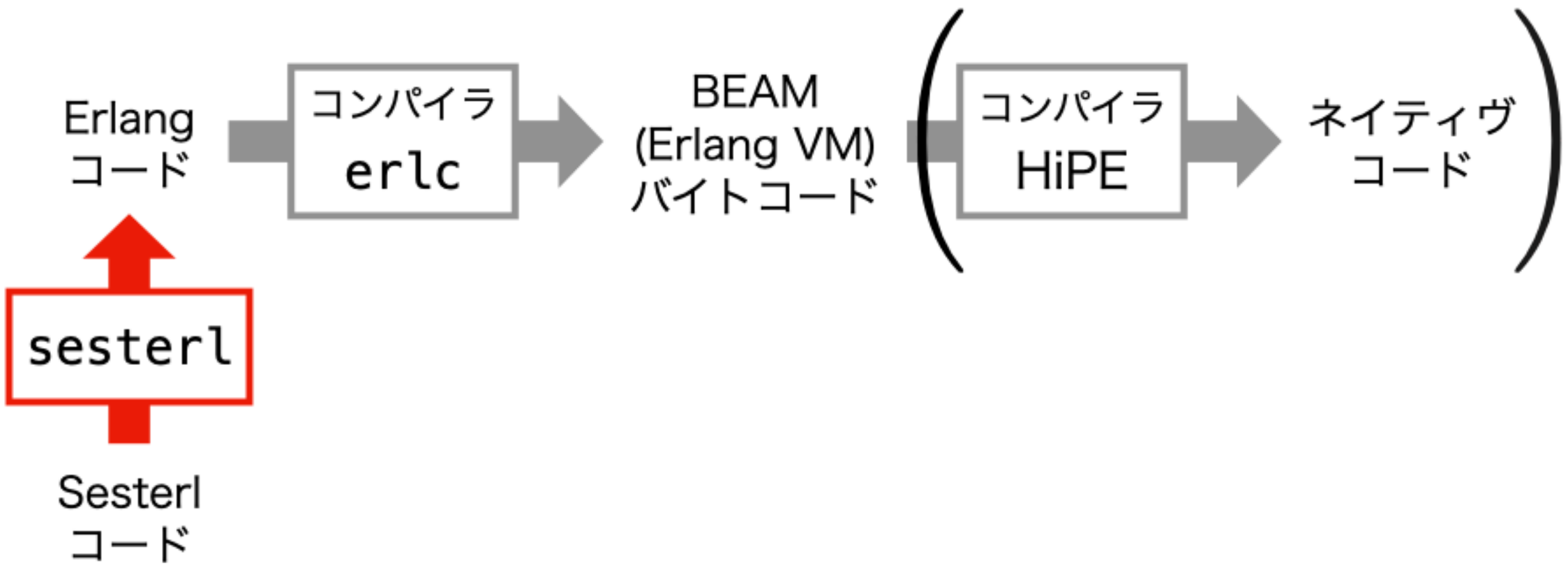
```
type option<$a> =
   | None
   | Some($a)

let 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` $\neq$ `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

- ~~ラベルつきオプション引数~~
- モジュールシステム
- Future Work
- まとめ

純粋/並行の区別 (1/2)

- モナドによって純粋/並行を区別 [Orchard & Yoshida 2016] [Fowler 2019]
- 型の構文： $\tau \quad ::= \quad \cdots \mid [\tau]\,\tau$
- $[\tau]\,\tau'$ 型の直観：

  メッセージを受信するなら τ 型のメッセージでなければならず,
  かつ最終的に τ' 型の値が求まるような並行処理の型

$$\frac{\Gamma, x_i : \tau \vdash e_i : [\tau]\,\tau' \quad (\text{for each } i \in \{1, \ldots, n\})}{\Gamma \vdash \texttt{receive}\ \ (x_i \texttt{ -> } e_i)_{i=1}^{n}\ \texttt{end} \ : \ [\tau]\,\tau'}$$

$$\frac{\Gamma \vdash e_0 : \tau_0}{\Gamma \vdash \texttt{return}(e_0) : [\tau]\,\tau_0}$$

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

純粋/並行の区別 (2/2)

- 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：

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

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

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

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

```
let main =
  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

▶ ラベルつきオプション引数

- モジュールシステム
- Future Work
- まとめ

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- ~~ラベルつきオプション引数~~

▶ **モジュールシステム**

- Future Work
- まとめ

モジュールシステム

- モジュールシステムとは：
  - いわゆるカプセル化を型システム的手法で実現する仕組みのひとつ
  - 例えば Standard ML や OCaml に備わっている
- 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
- **Static interpretation** [Elsman, Henriksen, Annenkov & Oancea 2018] という手法により部分評価してファンクタをコンパイル時に除去する処理も実装
  - ファンクタを使った実装を Erlang にコンパイルするのに実質必須だった

ファンクタの使用例

※Erlang/OTP ご存知の方向け：

gen_server はファンクタで
表現可能

- コールバック函数と
  それに関連する型を集めた
  モジュールを受け取る
- start_link, call, cast などを
  実装したモジュールを返す

右の gen_server 利用例 Cell：

整数 1 個を状態としてもち,
外部から参照・変更できる
プロセスを実装したモジュール

```
module Cell = struct
  module Callbacks = struct
    type state = int
    type init_arg = int
    type call_request = Get
    type cast_response = Got(int)
    type cast_msg = Set(int)
    let init(n : init_arg) = Ok(n)
    let handle_call(Get, n : state) =
      Ok(Reply(Got(n), n))
    let handle_cast(Set(m), n : state) =
      Ok(NoReply(m))
  end
  include GenServer.Make(Callbacks)

  let get_number(pid) =
    do response <- call(pid, Get) in
    case response of
    | Got(n) -> return(n)
    end

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

広告

ヤバイテックトーキョー vol. 5

F-ing modules を言語に実装する方法について記事を書きましたのでよろしければお求めください🙏

https://techbookfest.org/product/5131487676465152?productVariantID=612522130617139 2

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- ~~ラベルつきオプション引数~~
- モジュールシステム

▶ 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）
- Erlang のビルドツール **rebar3** との連携
- GADT
  - 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
- パッケージシステムの成熟
- 標準ライブラリの整備
- パターンマッチの網羅性検査

- 問題意識
- 基本的な言語設計
- 純粋計算/並行処理の区別
- ~~ラベルつきオプション引数~~
- モジュールシステム
- Future Work

▶ まとめ

まとめ

- ML 風の型つき AltErlang 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
    - ラベルつきオプショナル引数
    - ファンクタなどを備えた強力なモジュールシステム

参考文献

- 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.
- Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, **17**(6), pp. 844–895, 1995.
- 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.