Lessons from Erlang

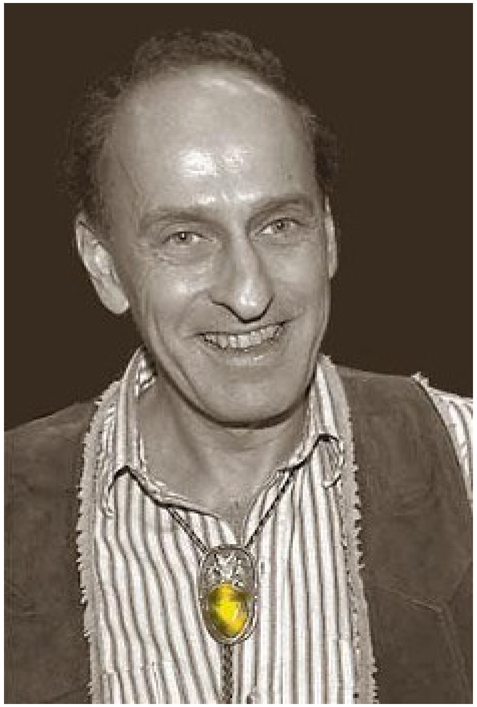
Simon Zelazny

LambdaCon 2015
2015-03-28, Bologna, Italy

Lessons from Whom?

- Born in 1983 in Poland. Older than Erlang by ~3 years.
- Studied Japanese at Warsaw U and Tokyo Gakugei U
- Got feet wet with web development and Ruby
- Adopted by a roving band of Haskell coders in 2011
  - Agile development, tight customer feedback loops
  - Greenfield projects, time for research
- Found a new home with Erlang Solutions in 2014
  - Dev methodologies differ customer-to-customer
  - Maintenance, integration, bug-fixes of large existing projects

Probably Not Related



(https://upload.wikimedia.org/wikipedia/en/f/fe/Rogerzelazny.JPG
)

The Design of Erlang

Original design requirements:

1) Concurrency
2) Soft real-time
3) Distributed
4) Hardware interaction
5) Large software systems
6) Complex functionality
7) Continuous operation
8) Quality requirements
9) Fault tolerance

The Design of Erlang

1) Concurrency

2) Distributed

3) Continuous operation

4) Fault tolerance

Erlang as an Operating System

- Built-in concurrency

```
-module(concurrency).
-export([run/0]).

churn() ->
    churn().

run() ->
    spawn(fun churn/0).
```

```
3> Churn1 = concurrency:run().
<0.44.0>
4> is_pid(Churn1).
true
```

Erlang as an Operating System

- Processes are isolated

```erlang
-module(isolation).
-export([run/0]).

crash() ->
    100 / 0.

run() ->
    spawn(fun crash/0),
    ok.
```

```
12> isolation:run().
ok
```

Erlang as an Operating System

- Processes communicate by passing messages

```
-module(message).
-export([run/0]).

echo(Parent) ->
    fun() -> Parent ! {echo_from, self()} end.

run() ->
    Child = spawn(echo(self())),
    receive
        {echo_from, Child} -> {ok, Child}
    end.
```

```
4> message:run().
{ok,<0.46.0>}
```

Erlang as an Operating System

- Errors can be promptly detected (and propagated)

```
-module(errors).
-export([run/0]).

crash() ->
    100 / 0.

run() ->
    spawn_monitor(fun crash/0),
    receive
        ErrorNotification -> ErrorNotification
    end.
```

```
12> errors:run().
{'DOWN',#Ref<0.0.0.264>,process,<0.71.0>,
        {badarith,[{errors,crash,0,
                           [{file,"errors.erl"},{line,5}]}]}}
```

Erlang as an Operating System

- Distribution is built-in

```
$ erl -sname yang -setcookie chocolate_chip

Eshell V6.3  (abort with ^G)
(yang@kos)1> nodes().
[]
(yang@kos)2> net_adm:ping('yin@kos').
pong
(yang@kos)3> nodes().
[yin@kos]
```

```
$ erl -sname yin -setcookie
chocolate_chip
Eshell V6.3  (abort with ^G)
(yin@kos)1> nodes().
[]



(yin@kos)2> nodes().
[yang@kos]
```

Erlang as an Operating System

- Distribution is built-in and supports all the goodies

```
$ erl -sname yang -setcookie chocolate_chip
Eshell V6.3  (abort with ^G)
(yang@kos)1> nodes().
[]
(yang@kos)2> net_adm:ping('yin@kos').
pong
(yang@kos)3> nodes().
[yin@kos]
(yang@kos)4> spawn('yin@kos',
                   fun errors:run/0).
<7060.45.0>

(yang@kos)5>
=ERROR REPORT==== 25-Mar-2015::23:42:09 ===
Error in process <0.46.0> on node 'yin@kos'
with exit value: {badarith,[{errors,crash,0,
[{file,"errors.erl"},{line,5}]}]}
```

```
$ erl -sname yin -setcookie
chocolate_chip
Eshell V6.3  (abort with ^G)
(yin@kos)1> nodes().
[]
(yin@kos)2> nodes().
[yang@kos]
```

Erlang as a Practical System

- Battle-tested

Erlang as a Practical System

- Battle-tested
- Opinionated

Erlang as a Practical System

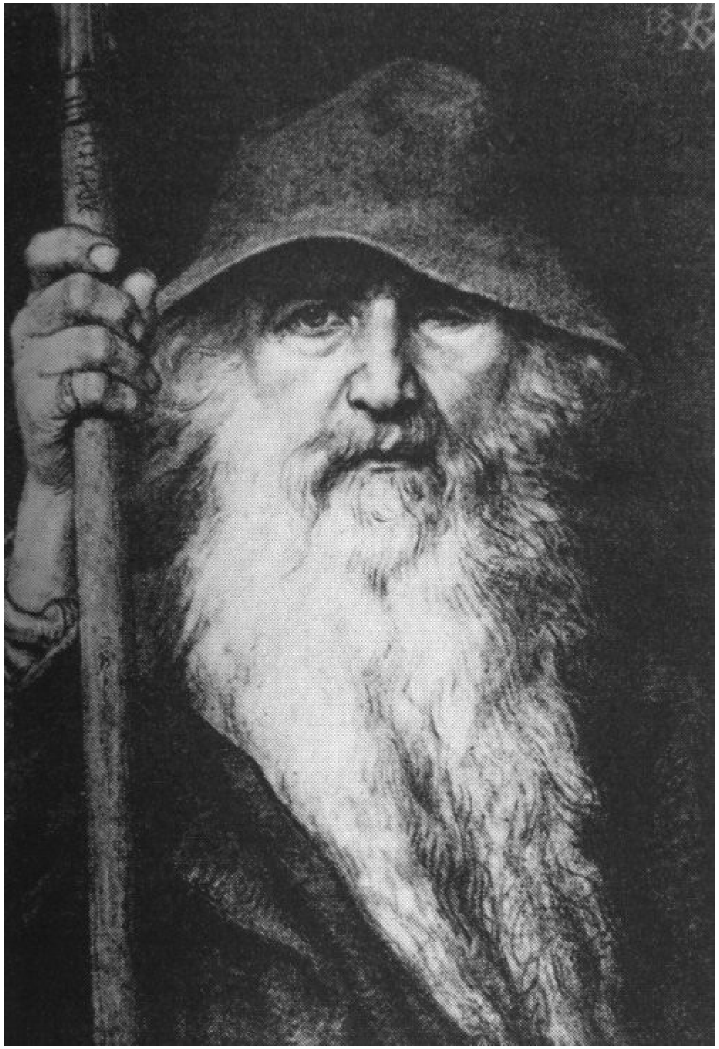
- Battle-tested
- Opinionated
- Inconsistent & quirky

Erlang as a Practical System

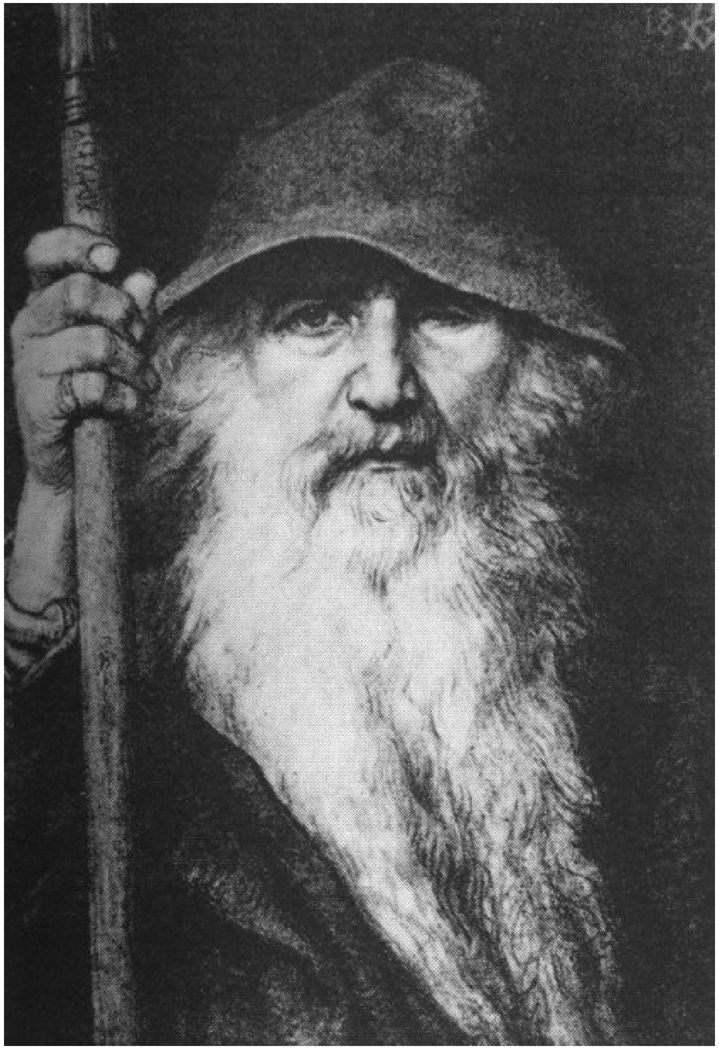
- Battle-tested
- Opinionated
- Inconsistent & quirky
- Historical baggage

Erlang as a Practical System

- Battle-tested
- Opinionated
- Inconsistent & quirky
- Historical baggage
- Wise



And now, for something completely different...



Lessons from Erlang: The Good

- OTP
- Supervision trees
- Introspection and debugging
- Distribution
- Hot code reload
- The Erlang shell

Lessons from Erlang: The Bad

- Records
- Cruft
- Flat namespaces
- Awkward higher-order programming

Lessons from Erlang: The Ugly

- The type system and related tooling
- User (programmer) experience
  - Visual clutter and verbosity
  - Comment culture and conventions
  - Wonky testing libraries
- Macros and parse transforms

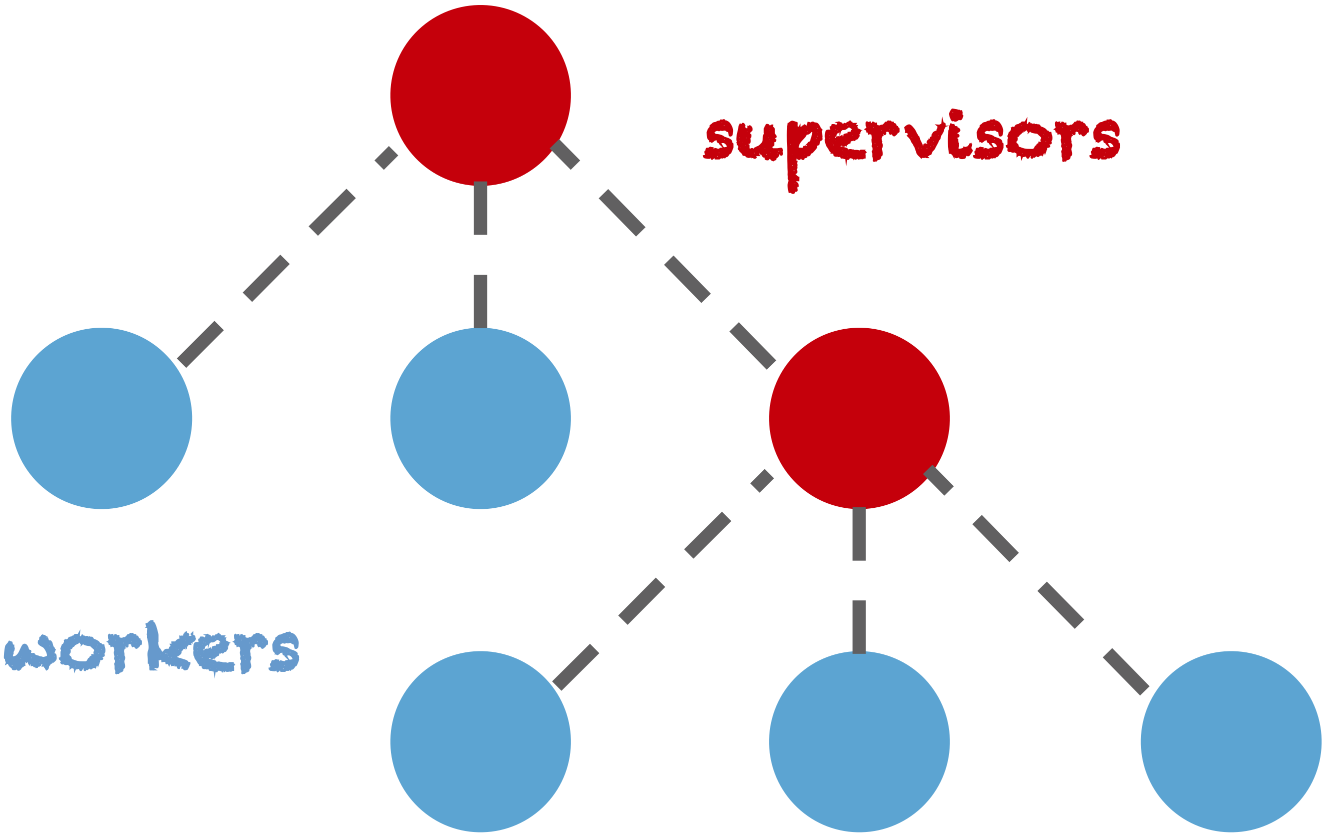
The Good

OTP

- A set of predefined behavioral patterns for consistent design
  - Client/server
  - Finite state machine
  - Event handler
  - Supervisor

- A methodology for managing software in the large and consistent operations & maintenance
  - Applications
  - Releases

Supervision Trees

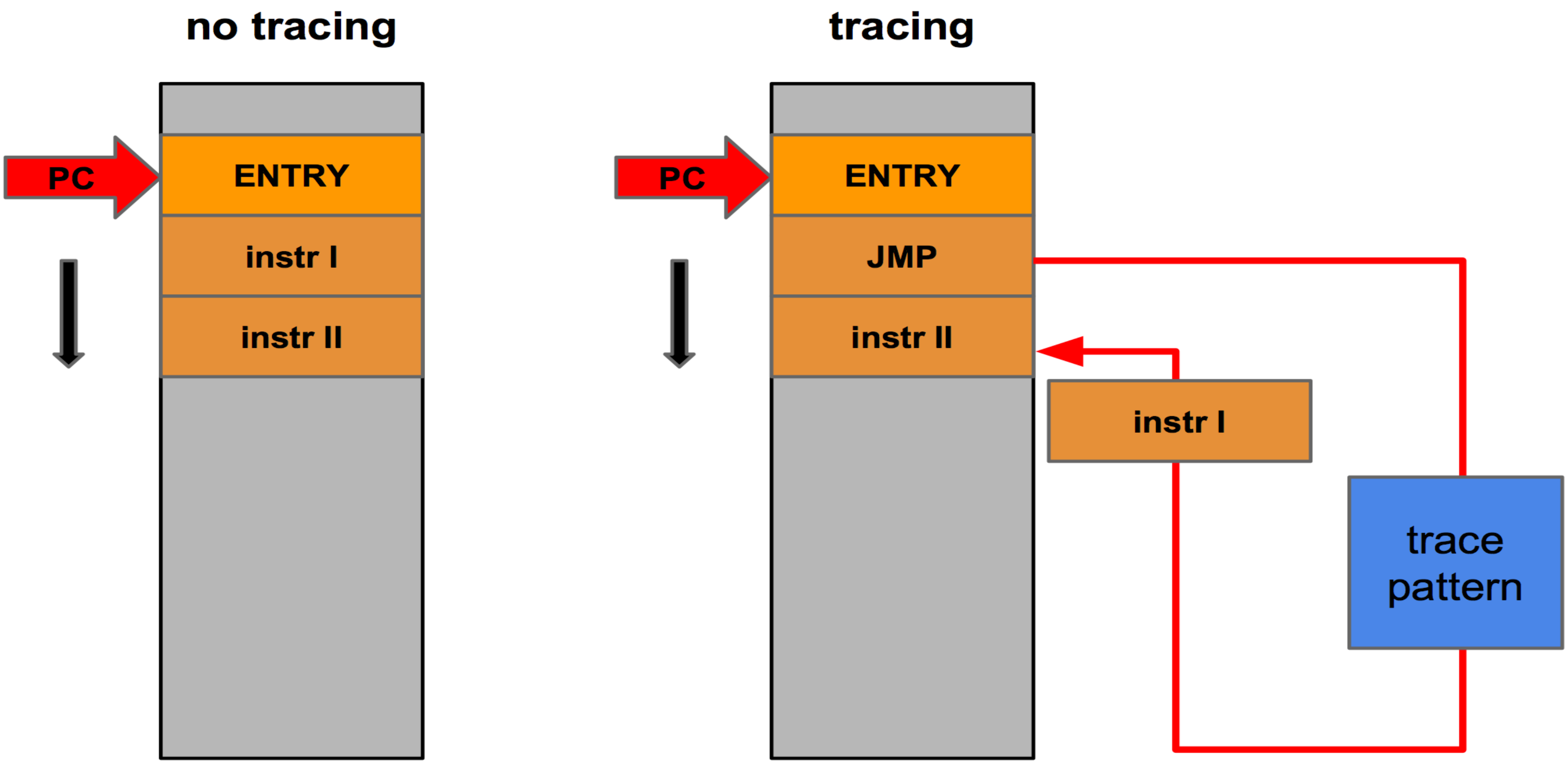


Supervision Trees

- Program for the correct case only
- Let it crash!
- Subsystems will organically return to their initial, stable state
- The system has a good chance of functioning despite (heisen-)bugs and failures

Introspection and Debugging

- Extensive (and dangerous!) tracing capabilities built into the VM (Image courtesy of Mats Cronqvist)



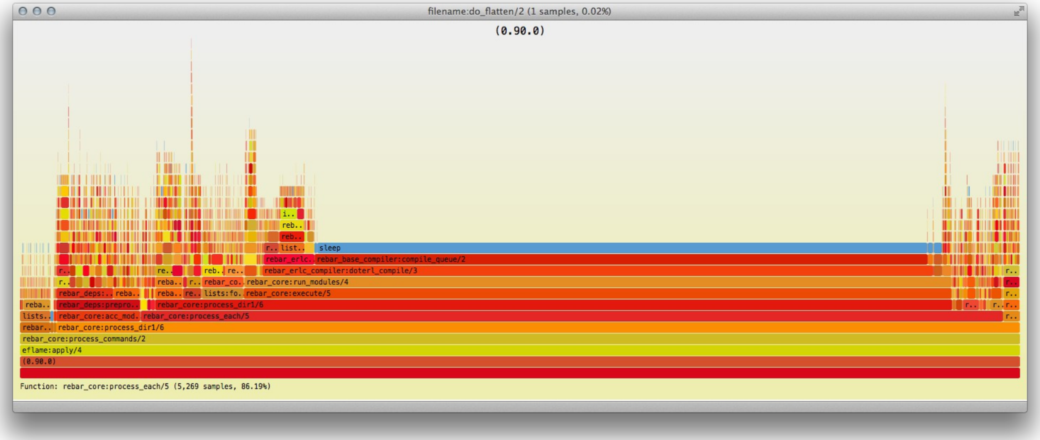
Introspection and Debugging

- Awesome tools built on top of the VM's tracing capabilities
  - Recon (http://ferd.github.io/recon/)

```
1> recon_trace:calls(
1>    {queue, '_',
fun([A,_]) when is_list(A); is_integer(A) andalso A > 1 ->
return_trace() end}, {10,100}).

13:24:21.324309 <0.38.0> queue:in(3, {[],[]})
13:24:21.371473 <0.38.0> queue:in/2 --> {[3],[]}
```

  - Eflame (https://github.com/proger/eflame)



filename:do_flatten/2 (1 samples, 0.02%)

Function: rebar_core:process_each/5 (5,269 samples, 86.19%)

Introspection and Debugging

- The Erlang runtime system gives lots of information on its inner workings
  - **erlang:process_info/1,2**

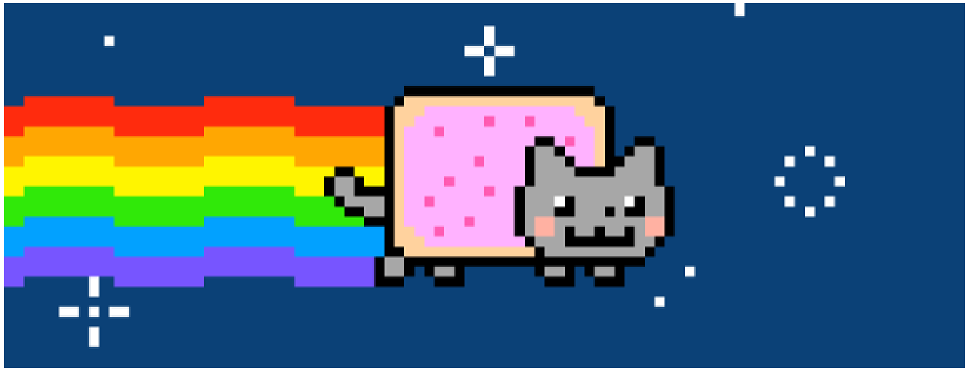
    Gives lots of information about a process: its message queue, links, monitors, garbage collections, etc.
  - **sys:get_status/1,2**

    Inspects and OTP-compliant process about its place in supervision tree and state.
  - **erlang:statistics/1**

    Returns various runtime info: io, scheduler utilization, gc size, etc.
  - **… and more!**

Hot Reload and the Erlang Shell

- Works like magic! Even on remote nodes!
- Compile, inspect and load new versions of modules into a running system
- Calls to updated functions will pick up the new code even in long-living processes*



The Bad

#records{}

- The syntax is OK, but records...
  - need to be defined in headers and included(!)
  - don't print nicely in the shell (field names are gone!)
  - work differently in the shell than inside modules
- Should record types be defined in headers as well?

```
2> rr("rec.hrl").
[album]
3> DSOTM = #album{year = 1973, artist = "Pink Floyd", title = "Dark Side of
the Moon"}.
#album{title = "Dark Side of the Moon", artist = "Pink Floyd",year = 1973}
4> io:format("~p~n", [DSOTM]).

{album,"Dark Side of the Moon","Pink Floyd",1973}
```

Cruft

- Strange interfaces

```
5> MyStuff = [{false, true}, false, {1, false}].
[{false,true},false,{1,false}]
6> lists:keyfind(false, 1, MyStuff).
{false,true}
7> lists:keyfind(true, 1, MyStuff).
false
```

- Placeholder-driven design (from the *erlang* man page:)

monitor(Type, Item) -> MonitorRef
The calling process starts monitoring Item which is an object of type Type.
Currently only processes can be monitored, i.e. the only allowed Type is process, but other types may be allowed in the future.

monitor_node(Node, Flag) -> true
...

Flat Namespaces

- No way to encapsulate internal business logic or models while keeping them in separate files
  - `-module(myapp_userservice_user_model).`
  - `-module(myapp_userservice_db_backend).`

- A rock and a hard place: qualified function calls vs. 80 column line length limit...

- Single, global namespace in the entire Erlang ecosystem
  - grab the short app/library names and squat them!

Awkward Higher-Order FP

- No **`let`**s
- No **`where`**s
- No partial application

  - Plug: (https://github.com/lavrin/pa)
- Verbose lambdas
- LISP-1-style function references: **`fun foo/X`** vs. **`foo`**
- No composition operator
- Incomplete list operation library
  - Shameless plug: (https://github.com/pzel/l)

- ...somewhat alleviated by pattern matching and list comprehensions.

The Ugly

The Type System & Type Tools

- Dialyzer, Typer, Xref and others work well, but they're hard to use and weakly integrated with the Erlang compiler.
- **Success Typing** is very cool, but it's not **Hindley-Milner**

$$\frac{}{A \cup \{x \mapsto \tau\} \vdash x : \tau,\ \emptyset} \qquad [\textsc{Var}]$$

$$\frac{A \vdash e_1 : \tau_1,\ C_1\ \ldots\ e_n : \tau_n,\ C_n}{A \vdash c(e_1, \ldots, e_n) : c(\tau_i, \ldots, \tau_n),\ C_1 \wedge \ldots \wedge C_n} \qquad [\textsc{Struct}]$$

$$\frac{A \vdash e_1 : \tau_1, C_1 \quad A \cup \{x \mapsto \tau_1\} \vdash e : \tau_2,\ C_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2,\ C_1 \wedge C_2} \qquad [\textsc{Let}]$$

$$\frac{A \cup \{x_i \mapsto \tau_i\} \vdash f_1 : \tau_1',\ C_1\ \ldots\ f_n : \tau_n',\ C_n \quad e : \tau,\ C}{A \vdash \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e : \tau, C_1 \wedge \ldots C_n \wedge C \wedge (\tau_1' = \tau_1) \wedge \ldots \wedge (\tau_n' = \tau_n)} \qquad [\textsc{Letrec}]$$

$$\frac{A \cup \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau_e, C}{A \vdash \texttt{fun}(x_1, \ldots, x_n) \to e : \tau,\ (\tau = ((\tau_1, \ldots, \tau_n) \to \tau_e\ \mathit{when}\ C))} \qquad [\textsc{Abs}]$$

$$\frac{A \vdash e_1 : \tau_1,\ C_1\ \ldots\ e_n : \tau_n,\ C_n}{\begin{array}{r} A \vdash e_1(e_2, \ldots, e_n) : \beta, (\tau_1 = (\alpha_2, \ldots, \alpha_n) \to \alpha) \wedge (\beta \subseteq \alpha) \wedge \\ (\tau_2 \subseteq \alpha_2) \wedge \ldots \wedge (\tau_n \subseteq \alpha_n) \wedge C_1 \wedge \ldots \wedge C_n \end{array}} \qquad [\textsc{App}]$$

$$\frac{A \vdash p : \tau,\ C_p \quad A \vdash g : \texttt{true},\ C_g}{A \vdash p\ \texttt{when}\ g : \tau,\ C_p \wedge C_g} \qquad [\textsc{Pat}]$$

$$\frac{\begin{array}{c} A \cup \{v \mapsto \tau_v | v \in Var(p_1)\} \vdash p_1 : \alpha_1,\ C_1^p,\ \ b_1 : \beta_1,\ C_1^b \\ \vdots \\ A \vdash e : \tau,\ C_e \quad A \cup \{v \mapsto \tau_v | v \in Var(p_n)\} \vdash p_n : \alpha_n,\ C_n^p,\ \ b_n : \beta_n,\ C_n^b \end{array}}{\begin{array}{r} A \vdash \texttt{case}\ e\ \texttt{of}\ p_1 \to b_1; \ldots p_n \to b_n\ \texttt{end} : \beta,\ \ C_e \wedge (C_1 \vee \ldots \vee C_n) \\ \text{where } C_i = ((\beta = \beta_i) \wedge (\tau_i = \alpha_i) \wedge C_i^p \wedge C_i^b) \end{array}} \qquad [\textsc{Case}]$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad [\texttt{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \qquad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \qquad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \qquad [\texttt{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \qquad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \qquad [\texttt{Gen}]$$

User (Programmer) Experience

- Tools produce too much output during normal (successful) operation
- Erlang heroes champion banner-style comments, instead of self-documenting code
- Common test hides your failing test results three our four pages deep inside a variably-named HTML log directory on your disk. Happy clicking
- Erlang programmers seem OK with these things! Must be Stockholm syndrome!

Macros and Parse Transforms

- Need to change the semantics of your code? Need more control over evaluation? Pick your poison:

- Macros
  - need to be included in header files if you want to resue them
  - ugly ?SYNTAX

- Parse transforms
  - throw off Erlang's awesome tracing tools
  - require an understanding of the Erlang AST
  - need to be compiled before the modules that use them

The End

The End

…questions?

Simon Zelazny
simon@manisola.net
https://github.com/pzel