# Concurrent Programming in Erlang

## The future of concurrent programming

Chad Hansen
Everett Morse
Jacob Fugal

# Outline

- History of Erlang
- Functional programming basics
- Syntax and semantics of Erlang
  - Pattern Matching
  - Example of Erlang program
- Concurrent components of Erlang
  - Example of concurrent Erlang program
- Considerations for message passing concurrency
- The future
  - Go
- Questions

# History of Erlang

- Developed at Ericsson telecom company in the 1980s.
- Named for Danish mathematician Agner Krarup Erlang (1878-1929).
- Designed for high reliability and distribution.
  - Fault tolerant
  - Soft real-time
  - Concurrent
  - Distributed
  - "Hot swapping"
- Motivated by problems which exhibit "natural" concurrency (e.g. telephone exchanges).
- Open sourced in 1998.

# Functional programming basics

- Declarative syntax.
  - Less about assignment and commands.
  - More about transformation via functions.
- Free from side effects (mostly).
  - Immutable data.
  - No shared state.
- Symbolic representation.
  - No explicit memory management.
  - Call by value (with optimizations).

# Syntax and semantics of Erlang

- Types
  - Numbers : `1, 1.0, etc.`
  - atoms : `'cat' =:= cat, true, false`
  - Lists : `[1, true, 'false'], "cat"`
  - Tuples : `{pay, 5.00}`
- Variables
  - immutable
- Functions : `->`
- Modules
- Operators
  - and, or, xor, not, andalso, orelse
  - +, -, *, /, div, rem
  - >, <, =<, >=, ==, /=, =:=, =/=

# Pattern Matching

- There is no assignment, only pattern matching.
- Variables are either *bound* or *free* .
- The *term* (right hand side of an =, parameter in function invocation, etc.) must contain only literals and bound variables.
- The *pattern* (left hand side of an =, parameter in function declaration, etc.) may contain bound or free variables.
- The "shape" of the term and pattern are compared
  - If the shape lines up, the match succeeds and any free variables in the pattern are bound to the corresponding values from the term.
  - If the shape is incompatible, the match fails.
- Use _ as a generic "always-free" placeholder.

# Example of Erlang program

```
-module(math1).
-export([double/1,times/2,factorial/1]).

double(X) ->
    times(X, 2).

times(X,N) ->
    X * N.

% factorial example.
factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

# Actor Model of Concurrency

- Lightweight processes (user-space, AKA threads or actors).
- No shared state (mostly).
- Asynchronous message passing.
- Mailboxes to buffer incoming messages.
- Mailbox processing via pattern matching.

No silver bullet:

- All the hazards of regularly concurrency are still possible.
- E.g. if process A and process B both know about and communicate with process C, then C itself is a form of shared state between A and B.

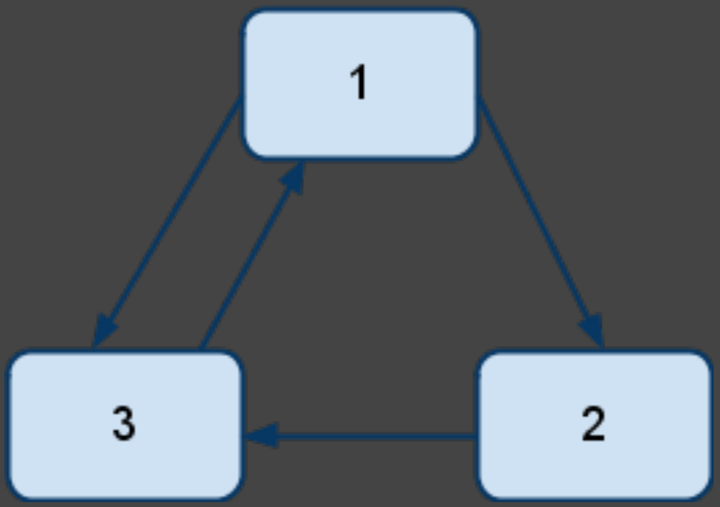
# Concurrent components of Erlang

Send a message to a process:

```
Pid ! Msg
```

Receive a message:

```
  receive
          Message1 ->
                ...;
          Message2 ->
                ...;
  end
```



# Concurrency example

```
-module(sequence).
-export([make_sequence/0,get_next/1,reset/1]).

make_sequence() ->
    spawn(fun() -> sequence_loop(0) end).

sequence_loop(N) ->
    receive
        {From, get_next} ->
            From ! {self(), N},
            sequence_loop(N+1);
        reset ->
            sequence_loop(0)
    end.

get_next(Sequence) ->
    Sequence ! {self(), get_next},
    receive
        {Sequence, N} -> N
    end.

reset(Sequence) ->
    Sequence ! reset.
```

# Considerations for message passing concurrency

- Processes are all light weight, so creating "servers" like the sequence example is not a problem.
- You can pass Process Ids in a message!
    - Register a callback / response channel
- The standard "OTP" library has three pre-built client-server models which can be extended:
    - Generic Server (gen_server)
    - Generic Finite State Machine (gen_fsm)
    - Generic Event Handler (gen_event)
- Use tail recursion in server loops
- Using the given generic templates helps avoid common pitfalls.

# Generic Server

- Standard request-response pattern
- Extra features:
  - Responses can be delayed or delegated
  - Calls can have optional timeouts
  - Client monitors server, receiving immediate notification of failure (not via timeout)

# Generic Finite State Machine

- Many concurrent algorithms can be modeled this way
- Calls update state, possible synchronous reply
- Callbacks in implementation return new state after each call

(I'll show a consensus protocol that could be implemented as a FSM.)

# Generic Event Handler

- Receives incoming events and passes them to any number of child handlers
- Each handler has it's own private state
- Handlers can be added/removed/changed
- Each handler typically cares about a few events to act on
- Works great for logging, monitoring, etc.

# Notes on Practical Use

- There are many libraries to use
- Light http web server
    - Mochiweb
- Database
    - Mnesia
    - MySQL connectors
- Web application framework
    - Nitrogen

Cons:

- Not as good at string manipulation, use Python/Perl/etc.
- Slower at number crunching

# Consensus Protocol

```
make_consensus() ->
spawn(fun() -> consensus(nil) end).

consensus(nil) ->
receive
{_, {propose, Value}} ->
                consensus(Value);
{From, get} ->
                From ! {self(), nil},
                consensus(nil);
{_, close} ->
                nil;  % end by not recursing
end;
consensus(Value) ->
receive
{_, {propose, _}} ->
                consensus(Value);
{From, get} ->
                From ! {self(), Value},
                consensus(Value);
{_, close} ->
                Value; % end by not recursing
end.
```

```
decide(Value, Consensus) ->
propose(Value, Consensus),
get_value(Consensus).


propose(Value, Consensus) ->
Consensus ! {self(), {propose, Value}}.

get_value(Consensus) ->
Consensus ! {self(), get},
receive
{_, Value} -> Value
after
1000 -> nil
end.
```

# The future

Erlang vs. other languages?

- Gaining visibility
- F# is a functional language building on .NET libraries
- Scala is a functional language building on Java
- Google's Go copies much from Erlang

Functional vs. Procedural

- Arguments that functional programming may move to main stream
- Erlang's benefits from:
  - Actor model - light weight communicating processes
  - No shared state
- Go is procedural, but copies the same concurrency features.
- Note: LISP-based languages confuse many people with parenthesis bloat. Erlang, F#, Scala, etc. do not.

# Google's Go

- Imitation is the sincerest form of flattery
- Uses "Communicating Sequential Processes" model, similar to Erlang's Actor model
- Uses a channel syntax for message passing, code easily ported
- Also: Copied the "_" variable, module usage, io:format -> fmt.Printf with similar options

Thus:

- Go's developers must have used and considered Erlang
- Erlang's methods are useful and will be copied

# Review

- History of Erlang
- Functional programming basics
- Syntax and semantics of Erlang
    - Example of Erlang program
- Concurrent components of Erlang
    - Example of concurrent Erlang program
- Considerations for message passing concurrency
- The future
    - Go
- Questions

**Questions?**

# Want more? Check out these sources

- Erlang for Concurrent Programming - http://queue.acm.org/detail.cfm?id=1454463
- Learn you some Erlang - http://learnyousomeerlang.com/content
- Erlang.org - http://www.erlang.org/doc/
- Concurrent Programming in ERLANG, second edition. Joe Armstrong, et al. Prentice Hall 1996 -
  Part 1 free: http://www.erlang.org/download/erlang-book-part1.pdf
- "Installing Erlang and a few libraries on Mac OS X" - http://medevyoujane.com/blog/2008/8/6/installing-erlang-and-a-few-libraries-on-mac-os-x.html