Message-Passing Concurrency in Erlang

M. Ahrens and N. Ramsey, for COMP 250RTS

November 8, 2017

Erlang in context

Condensed from Wikipedia[1]: Erlang was designed to improve applications in telephony. It was first implemented in Prolog and was influenced by the programming language PLEX used in earlier Ericsson exchanges. By 1988 Erlang had proven that it was suitable for prototyping telephone exchanges, but the Prolog interpreter was about 40 times too slow for production. In 1992 work began on the BEAM virtual machine, which compiles Erlang to C.

Erlang offers superior linguistic support for distributed programming, especially recovery from failure. Erlang programmers routinely build reliable systems from unreliable components.

Getting started

Erlang borrows much from Prolog. As in Prolog, and similar to Scheme, Values are atoms, number, strings, applications, and lists. An initial capital letter identifies a variable, and an initial lowercase letter identifies an atom. There is no static type system, and there are no algebraic data types as such, but the role of value constructor is played by atoms.

Erlang basics, with suggested reading

The free book *Learn you some Erlang* is a quick read, but it leaves a lot to be desired. To prepare for class, quickly visit these sections:

- The shell[2] shows `erl`, the Erlang shell. You can type expressions, and you can load file `filename.erl` with `c(filename).` Each expression is terminated with a dot.
- An Erlang source file doesn't just have definitions. It also needs a *module declaration*[3].
  - To make the file a module, `-module(filename).`
  - To make functions public, you need a `-export` directive, which lists exported functions by name and arity. (Like Prolog, Erlang distinguishes functions by the combination of name and arity.)

    Example:

    ```
    -export([main/0,print/1])
    ```

- Erlang's values[4] are roughly those of Prolog, plus functions:
  - numbers
  - variables (e.g., `X`)
  - atoms (e.g., symbols like `ok` and function or module names like `main`)
  - booleans `true` and `false`
  - tuples (e.g., `{1,2,true}`)
  - lists
    - `[1,2,3,4,5]`
    - Prolog head | tail syntax: `[H | T] = [1,2,3]`
    - strings (e.g., `"hello"`) are sugar for lists of binary data
    - binary-data strings (e.g., `<<255,255,255>>`)
    - list comprehensions (e.g., `[2 * N | N <- [1,2,3,4,5]]`)
- Functions with pattern matching[5]:
  - Function Declarations

    ```
    funname(arg1,arg2,...) -> e1,e2,e3.
    ```

  - Declarative pattern matching

    ```
    fib(0) -> 1.
    fib(1) -> 1.
    fib(X) -> fib(X-1) + fib(X-2).
    ```

  - Erlang else has if-else, cases, and guards.

[1] https://en.wikipedia.org/wiki/Erlang_(programming_language)
[2] http://learnyousomeerlang.com/starting-out#the-shell
[3] http://learnyousomeerlang.com/modules#module-declaration
[4] http://learnyousomeerlang.com/starting-out-for-real
[5] http://learnyousomeerlang.com/syntax-in-functions

Erlang's concurrency summarized

From Armstrong's PhD thesis, via Carlo Furia at Chalmers[6]:

- "Processes" (threads) are dirt cheap.
- Each process has a name ("PID").
- If you know the name, you can send it a message.
- Messages are delivered *asynchronously*.
- Exchange of messages is the only means of synchronizing two threads.

This model extends easily to parallel and distributed systems—though when we look at the implementation, we will see some artifacts from the days when Erlang ran only on a uniprocessor.

Process basics

More from Furia, about

- `Pid = spawn (Module, Function, Arglist).`
- `Pid ! Message` adds message to recipient's *mailbox* (asynchronous with recipient)
- `process_info(Pid, message_queue_len)` gives population of mailbox
- Receipt uses pattern matching—if there is no match, it *blocks*

```
receive
  P1 when C1 -> E1;
   ...
  Pn when Cn -> En
end
```

Receipt can find *any* matching message (order doesn't matter).

More on these from *Learn You...*[7].

Registered processes

From Furia:

```
register(Name, Pid)

unregister(Name)
```

[6] http://www.cse.chalmers.se/edu/year/2016/course/TDA383_LP3/files/lectures/Lecture08-message-passing.pdf

[7] http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#thanks-for-all-the-fish

Discussion questions

The model

(1) What is Erlang's model for concurrency?

  (a) How are threads created and destroyed?
  (b) What determines when a thread runs?
  (c) How do threads communicate?
  (d) How are concurrent computations synchronized?

(2) Along the same dimensions, compare Erlang's concurrency with Lua's concurrency (coroutines using `create`, `resume`, and `yield`).

(3) Along the same dimensions, compare Erlang's concurrency with Gord Cormack's micro-kernel for concurrency in C (`emit`, `absorb`, `p`, `v`, `talk`, and `listen`).

(4) In class, we won't compare these other systems with Cilk, which is more about parallelism than concurrency. But it's worth thinking about.

Using the model

One classic implementation of a semaphore is a process that receives messages. In a synchronous system, just sending messages is enough, but in an asynchronous system, each client must communicate with the semaphore to know when its messages have been received and acted on.

(5) Using Cormack's work as a model, design and implement counting semaphores for Erlang. Define function `sem/1` (create a new semaphore) as well as `p` and `v`.

Recall that a semaphore is either a natural number or a list of waiting processes.

A process's own PID (and therefore its address for sending messages) can be obtained by calling the Erlang builtin `self()`.

(6) Given Erlang's primitives, how would you implement Lua coroutines? We'll be implementing coroutines to discuss on Monday, so this is our chance to get a head start on the design.

Problem to work on outside of class

(7) We'll provide implementations of `cofringe` and `coeqfringe` written in Erlang. You'll use Erlang to write the coroutine operations `create`, `yield`, and `resume`. We'll discuss the results Monday.