Anonymity in Erlang

Mayer Goldberg & Guy Wiener

Abstract

Generally speaking, servers and clients in Erlang are implemented as named functions in named modules. Similarly, processes communicate via messages that have a statically-known structure, and specifically, with static tags, that serve as the "names" of the messages. This exposes a great deal of information about an Erlang application: The names of the modules, the name of the entry-point functions within the module, the "names" of the messages between the server and the client, etc.

In this work, we show how higher-order functions, and some well-studied techniques from functional programming, can be used to obtain anonymity of servers and messages.

1 Introduction

The basic looping construct in Erlang is the tail-recursive function call. Erlang compiles these into code that is as efficient as `while`-loops in other programming languages.

Servers in Erlang are typically written using recursive functions [2, Chapter 8.2] and [3, Example 4-1]. The server is implemented as a function that receives a message, and as long as the server *continues* after receiving the message, the server will be re-invoked by calling the respective function tail-recursively.

Invoking a function on an Erlang node involves calling one of the `spawn` functions with the node (optional), the module name, the function name, and the list of its arguments. It is also possible to call `spawn` by passing it a closure as an argument. Any global name that is evaluated by this function must be defined on the node on which the spawned process will be running. If the name is undefined a run-time error will occur.

The fact that the server is recursive exposes several things about the underlying architecture:

- The name of the module in which the server appears needs to be known to the client.
- The client must also know the name of the server function and its arity.
- There must be, on the server, a mounted file system. If the module file is located on that file system, then the Erlang system must have read privileges to it. If the module file is located only on the client, the Erlang system must have write privileges to it.

There are situations in which these constraints are undesirable. Erlang is currently available for many platforms, including thin clients and mobile devices. Forcing the client and the server to share files in advance prevents users from taking advantage of available Erlang nodes on resource-restricted platforms. Even if accessing the file system is permitted, it places a constraint on general-purpose servers. Instead of having the client hold the software that it wants to send to the general-purpose server, it forces the client to first update the server and only then run the new software on the server. Exposing the structure of the server module is also not always desired, since that makes the server writer more focused on the public functions of the module, rather then on the message protocol.

2 Replacing recursion with self-application

Any recursive call can be replaced by a call to a function reference that is passed as an argument. Consider the following example of the factorial function, given in the ubiquitous C programming language:

Recursive version:

```
int fact(int n) {
  return (n == 0) ?
         1 :
         n * fact(n - 1); }
```

Used as:

```
int n_fact = fact(n);
```

Self-applicative version:

```
int fact(void *f, int n) {
  reutrn (n == 0) ?
         1 :
         n * ((int (*)(void *, int))f)(f, n - 1); }
```

Used as:

```
int n_fact = fact(&fact, n);
```

Notice that the self-applicative version computes the factorial function only when the function is applied to itself, along with an integer argument. If we were to code the self-applicative version of factorial in Erlang, we would get:

```
fact(F, 0) -> 1;
fact(F, N) -> N * (F(F, N - 1)).
```

which can be invoked as follows:

```
fact(fun fact/2, N).
```

Note that the self-applicative function `fact` is not recursive. It can be written entirely using anonymous functions:

```
Fact =
fun (N) ->
  (fun (F) -> F(F, N) end)
  (fun (F, N) ->
    case N of
     0 -> 1;
     _ -> N * F(F, N - 1)
    end
   end)
end.
```

and this is how it can be invoked:

```
> Fact(5).
120
> Fact(7).
5040
```

Writing such functions can be simplified by slightly changing the interface of such functions, and abstracting out a general-purpose “recursion-maker”, known in the functional programming community as a “fixed-point combinator” [4, Page 178]:

```
Y1 =
fun (F) ->
 (fun (X) -> X(X) end)
 (fun (M) ->
   F (fun (Arg) ->
       (M(M))(Arg) end)
  end)
end.
```

And here is how `Y1` could be used:

To define factorial:

```
Factorial =
Y1 (fun (Fact) ->
      fun (N) ->
        case N of
          0 -> 1;
          _ -> N * Fact(N - 1)
        end
      end
    end).
```

To define Fibonacci:

```
Fibonacci =
Y1 (fun (Fib) ->
      fun (N) ->
        case N of
          0 -> 0;
          1 -> 1;
          _ -> Fib(N - 1) +
               Fib(N - 2)
        end
      end
    end).
```

We can now use our functions as follows:

```
> Factorial(5).
120
> Factorial(7).
5040
```

```
> Fibonacci(10).
55
> Fibonacci(20).
6765
```

In languages that support variadic functions (functions that take any number of arguments), for example, Scheme, Python, etc., it is possible to extend `Y1` so that it can be used to define any number of mutually-recursive functions, each taking any number of arguments [6, 7]. Because Erlang lacks support for defining variadic functions, we shall not pursue this direction here, but rather encode our functions using self-application directly.

3 Anonymous servers

As we have seen, it is possible to rewrite recursive functions, so that the recursive call is replaced by self-application. This allows recursive functions to be written anonymously, that is without using a globally-defined name for the function. Since a server in Erlang is a specific kind of recursive function, this same rewriting strategy can be used to create anonymous Erlang servers.

Here is a toy server for returning successive Fibonacci numbers:

```
FibonacciServer =
(fun (X) -> (X(X))(0, 1) end)
(fun (M) ->
  fun (N1, N2) ->
   fun () ->
    receive
     {fib, Pid} ->
       Pid ! {fib, N1},
       ((M(M))(N2, N1+N2))() ;
     restart -> ((M(M))(0, 1))() ;
     done -> ok
     end
   end
  end
 end).
```

The server answers 3 messages:

- {`fib`, *`Pid`*}, to have the next number in the Fibonacci sequence sent to the process ID *`Pid`*.
- `restart`, to restart the Fibonacci server.
- `done`, to stop the Fibonacci server.

This is the entire code for the server. Any client can pass this code onto any server in its cluster, via the `spawn` command. Here is how it can be used:

```
(one@erlang.edu)26> P = spawn('two@erlang.edu', FibonacciServer).
<5574.42.0>
(one@erlang.edu)27> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)28> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)29> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)30> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)31> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)32> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
ok
(one@erlang.edu)33> P ! restart.
restart
(one@erlang.edu)34> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)35> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)36> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)37> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)38> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)39> P ! {fib, self()}.
{fib,<0.54.0>}
(one@erlang.edu)40> P ! done.
done
(one@erlang.edu)41> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
Shell got {fib,5}
ok
```

Notice that we do not know anything about the node on which the server is running other than its address: No paths, module names, function names, etc.

4 Anonymous messages

Now that we have anonymous functions, we move on to anonymous messages. Messages typically have a static structure to them, they are usually ordered tuples the first tuple of which is an atom that serves as a tag that identifies the type of message.

For many purposes it is desirable that message tags be privately shared between a client and a server, and not accessible otherwise. This is quite easily achieved as an extension to our anonymous server. For private tags in the following examples, we chose 20-digit long pseudo-randomly generated large integers. In a realistic application, we might use larger integers, or might prefer pseudo-randomly generated atoms of great length.

In the following example, we modified the Fibonacci server, to define a server-client pair in which the client asks for the next Fibonacci number using a message that is tagged via a secret tag, generated at run-time, and shared only between this specific server and client. The code could be used to create any number of such server-client pairs, each pair sharing its own secret pseudo-random tag. We left out the definition for the pseudo-random number generator `Random/0`, which just calls `random:uniform/0` several times, and creates an integer of the right length.

```
ServerAndClient  =
fun () ->
 (fun (Mfib) ->
   {(fun (X) -> (X(X)) (0, 1) end)
      (fun (M) ->
        fun (N1, N2) ->
         fun () ->
          receive
           {Mfib, Pid} ->
             Pid ! {fib, N1},
             ((M(M))(N2, N1+N2))() ;
           restart -> ((M(M))(0, 1))() ;
           done -> ok
         end
        end
       end
     end),
    (fun (Pid) -> Pid ! {Mfib, self()}, ok end)}
   end)
 (Random())
end.
```

The following interaction shows how to define a server-client pair, how to spawn the server, how the client communicates with it, and what messages are received in response.

```
6> {Server, Client} = ServerAndClient().
{#Fun<erl_eval.20.117942162>,#Fun<erl_eval.6.35866844>}
7> Pid = spawn(Server).
<0.39.0>
8> Client(Pid).
ok
9> Client(Pid).
ok
   ...
14> Client(Pid).
ok
15> flush().
Shell got {fib,0}
Shell got {fib,1}
Shell got {fib,1}
Shell got {fib,2}
Shell got {fib,3}
Shell got {fib,5}
Shell got {fib,8}
ok
```

The value of `Mfib` is a 20-digit integer, and is unique to the client-server pair. We could just have easily have picked a 40-digit, or 100-digit integer, rendering impractical any attempt to arrive at the tag, either by guessing, or by systematically trying every integer.

In fact, we can make the tag even more secure, by having the client and server re-select a new tag every so often. An unauthorized client that might try to connect to the server by trying out numbers sequentially would be able to conclude nothing from past failures.

Here is a function for creating server-client pairs that re-select the tag after each use. Each call to `ServerClientAndInitialTag` would return a new triple of server, client, and an initial tag by which both the server and the client are synchronized:

```
ServerClientAndInitialTag =
fun () ->
 (fun (Mfib) ->
   {(fun (X) -> (X(X)) (0, 1, Mfib) end)
      (fun (M) ->
        fun (N1, N2, MfibMsg) ->
         fun () ->
          receive
           {MfibMsg, Pid} ->
             (fun (NewMfibMsg) ->
               Pid ! {fib, N1, NewMfibMsg},
               ((M(M))(N2, N1+N2, NewMfibMsg))()
              end)
             (Random());
           restart -> ((M(M))(0, 1))() ;
           done -> ok
         end
        end
       end
     end),
    (fun (Pid, Msg) -> Pid ! {Msg, self()}, ok end),
    Mfib}
  end)
  (Random())
end.
```

The following interaction shows how to define a server-client pair, what is the value of the first tag, how to spawn the server, how the client communicates with it, and what messages are received in response, and how the values of subsequent tags change according to the value of the responses from the server.

```
5> {Server, Client, InitialTag} = ServerClientAndInitialTag().
{#Fun<erl_eval.20.117942162>,#Fun<erl_eval.12.35291978>,
 684685090982776576}
6> Pid = spawn(Server).
<0.38.0>
7> Client(Pid, 684685090982776576).
ok
8> flush().
Shell got {fib,0,9230532871182784512}
ok
9> Client(Pid, 9230532871182784512).
ok
10> flush().
```

```
Shell got {fib,1,31133272937120710656}
ok
11> Client(Pid, 31133272937120710656).
ok
12> flush().
Shell got {fib,1,59651150244340162560}
ok
13> Client(Pid, 59651150244340162560).
ok
14> flush().
Shell got {fib,2,55825795815156547584}
ok
15> Client(Pid, 55825795815156547584).
ok
16> flush().
Shell got {fib,3,56212452643441508352}
ok
17> Client(Pid, 56212452643441508352).
ok
18> flush().
Shell got {fib,5,57882298363287674880}
ok
```

As can be seen, each message sent to the server has its own message tag that is synchronized between the server and the client. Each message sent from the server to the client contains, in addition to the `fib` tag and the next value in the Fibonacci sequence, the message tag to be used by the client in the subsequent request.

Any realistic client would need, of course, to receive the messages sent back by the server, and especially in this last example, where the client would be unable to communicate with the server unless it obtained the name of the subsequent message tag. For brevity, however, we presented only the simplest server-client pairs that use anonymous messages to communicate.

5 Discussion

Self-application has its origins in the λ-calculus and combinatory logic, where it is central to defining recursive functions. Functional programming languages, and especially those that are dynamically-typed, can use self-application in place of recursion. Exercises related to self-application and recursion are common in functional programming courses. For example, *Structure and Interpretation of Computer Programs* [1, Section 4.1.7, Page 393], and *The Little LISPer* [5, Chapter 9, Page 171].

In Erlang, self-application is more than just a programming exercise. While recursive functions cannot be passed between nodes, it is possible to pass functions that use self-application in place of recursion. We have thus been able to pass complete [albeit, small] servers among nodes. This has been done without requiring any access to the file system on the server host.

Once we have an entire server as a higher-order, non-recursive function that can be passed between nodes, it is straightforward to abstract over the message tags, and create server-client pairs that have their own, private message tags. For some added privacy, we can even have the message tags change between message calls.

In the servers we demonstrated, the message tags are selected pseudo-randomly during run-time. This is significant, because at no point does the source code contain the message tags. The message tags are private even if the source code is available.

If the pseudo-random number generator in these examples is replaced by a true, hardware-based, random number generator, a formidable level of privacy should be achieved.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company, Second edition, 1996.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly, June 2009.

[4] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[5] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. Science Research Associates, Inc, 1986.

[6] Mayer Goldberg. A Variadic Extension of Curry's Fixed-Point Combinator. In Olin Shivers, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 69–78, October 2002.

[7] Christian Queinnec. *LISP In Small Pieces*. Cambridge University Press, 1996.