Extracted from:

Programming Erlang

Software for a Concurrent World

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong

Chapter 8

Concurrent Programming

In this chapter, we'll be talking about *processes*. These are small self-contained virtual machines that can evaluate Erlang functions.

I'm sure you've met processes before, but only in the context of operating systems.

In Erlang, processes belong to the programming language and NOT the operating system.

In Erlang:

- Creating and destroying processes is very fast.
- Sending messages between processes is very fast.
- Processes behave the same way on all operating systems.
- We can have very large numbers of processes.
- Processes share no memory and are completely independent.
- The only way for processes to interact is through message passing.

For these reasons Erlang is sometimes called a *pure message passing language*.

If you haven't programmed with processes before, you might have heard rumors that it is rather difficult. You've probably heard horror stories of memory violations, race conditions, shared-memory corruption, and the like. In Erlang, programming with processes is easy. It just needs three new primitives: **spawn**, **send**, and **receive**.

8.1 The Concurrency Primitives

Everything we've learned about sequential programming is still true for concurrent programming. All we have to do is to add the following primitives:

Pid = spawn(Fun)

Creates a new concurrent process that evaluates Fun. The new process runs in parallel with the caller. spawn returns a Pid (short for *process identifier*). You can use Pid to send messages to the process.

Pid ! Message

Sends Message to the process with identifier Pid. Message sending is asynchronous. The sender does not wait but continues with what it was doing. ! is called the *send* operator.

Pid ! M is defined to be M—the message sending primitive ! returns the message itself. Because of this, Pid1 ! Pid2 ! ... ! M means send the message M to all the processes Pid1, Pid2, and so on.

receive ... end

Receives a message that has been sent to a process. It has the following syntax:

```
receive
   Pattern1 [when Guard1] ->
       Expressions1;
   Pattern2 [when Guard2] ->
       Expressions2;
   ...
end
```

When a message arrives at the process, the system tries to match it against Pattern1 (with possible guard Guard1); if this succeeds, it evaluates Expressions1. If the first pattern does not match, it tries Pattern2, and so on. If none of the patterns matches, the message is saved for later processing, and the process waits for the next message. This is described in more detail in Section 8.6, *Selective Receive*, on page 155.

The patterns and guards used in a receive statement have exactly the same syntactic form and meaning as the patterns and guards that we use when we define a function.

8.2 A Simple Example

Remember how we wrote the area/1 function in Section 3.1, *Modules*, on page 45? Just to remind you, the code that defined the function looked like this:

Download geometry.erl

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R.
```

Now we'll rewrite the same function as a *process*:

Download area_server0.erl

```
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {circle, R} ->
            io:format("Area of circle is ~p~n", [3.14159 * R * R]),
            loop();
        Other ->
            io:format("I don't know what the area of a ~p is ~n",[Other]),
            loop()
    end.
```

We can create a process that evaluates loop/0 in the shell:

```
1> Pid = spawn(fun area_server0:loop/0).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {circle, 23}.
Area of circle is 1661.90
{circle,23}
4> Pid ! {triangle,2,4,5}.
I don't know what the area of a {triangle,2,4,5} is
{triangle,2,4,5}
```

What happened here? In line 1 we created a new parallel process. spawn(Fun) creates a parallel process that evaluates Fun; it returns Pid, which is printed as <0.36.0>.

In line 2 we sent a message to the process. This message matches the first pattern in the receive statement in loop/0:

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

Having received a message, the process prints the area of the rectangle. Finally, the shell prints {rectangle, 6, 10}. This is because the value of Pid ! Msg is defined to be Msg. If we send the process a message that it doesn't understand, it prints a warning. This is performed by the Other ->... code in the **receive** statement.

8.3 Client-Server—An Introduction

Client-server architectures are central to Erlang. Traditionally, client-server architectures have involved a network that separates a client from a server. Most often there are multiple instances of the client and a single server. The word *server* often conjures up a mental image of some rather heavyweight software running on a specialized machine.

In our case, a much lighter-weight mechanism is involved. The client and server in a client-server architecture are separate processes, and normal Erlang message passing is used for communication between the client and the server. Both client and server can run on the same machine or on two different machines.

The words *client* and *server* refer to the roles that these two processes have; the client always initiates a computation by sending a *request* to the server. The server computes a reply and sends a *response* to the client.

Let's write our first client-server application. We'll start by making some small changes to the program we wrote in the previous section.

In the previous program, all that we needed was to send a request to a process that received and printed that request. Now, what we want to do is send a response to the process that sent the original request. The trouble is we do not know to whom to send the response. To send a response, the client has to include an address to which the server can reply. This is like sending a letter to somebody—if you want to get a reply, you had better include your address in the letter!

So, the sender must include a reply address. This can be done by changing this:

```
Pid ! {rectangle, 6, 10}
```

to the following:

```
Pid ! {self(),{rectangle, 6, 10}}
```

self() is the PID of the client process.

To respond to the request, we have to change the code that receives the requests from this:

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

to the following:

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        ...
```

Note how we now send the result of our calculation back to the process identified by the From parameter. Because the client set this parameter to its own process ID, it will receive the result.

The process that sends the initial request is usually called a *client*. The process that receives the request and sends a response is called a *server*.

Finally, we add a small utility function called rpc (short for *remote procedure call*) that encapsulates sending a request to a server and waiting for a response:

Download area_server1.erl

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
```

Putting all of this together, we get the following:

Download area_server1.erl

```
-module(area_server1).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        {From, {circle, R}} ->
            From !  3.14159 * R * R,
            loop();
        {From, Other} ->
            From ! {error,Other},
            loop()
    end.
```

We can experiment with this in the shell:

```
1>  Pid = spawn(fun area_server1:loop/0).
<0.36.0>
2>  area_server1:rpc(Pid, {rectangle,6,8}).
48
3>  area_server1:rpc(Pid, {circle,6}).
113.097
4>  area_server1:rpc(Pid, socks).
{error,socks}
```

There's a slight problem with this code. In the function rpc/2, we send a request to the server and then wait for a response. *But we do not wait for a response from the server*; we wait for any message. If some other process sends the client a message while it is waiting for a response from the server, it will misinterpret this message as a response from the server. We can correct this by changing the form of the receive statement to this:

```
loop() ->
    receive
        {From, ...} ->
            From ! {self(), ...}
            loop()
        ...
```

and by changing rpc to the following:

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

How does this work? When we have entered the rpc function, Pid is bound to some value, so in the pattern {Pid, Response}, Pid is bound, and Response is unbound. This pattern will match only a message containing a two-tuple[1] where the first element is Pid. All other messages will be queued. (**receive** provides what is called *selective receive*, which I'll describe after this section.)

With this change, we get the following:

Download area_server2.erl

```
-module(area_server2).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

This works as expected:

```
1> Pid = spawn(fun area_server2:loop/0).
<0.37.0>
3> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

1. N-tuple means a tuple of size N, so two-tuple is a tuple of size 2.

There's one final improvement we can make. We can *hide* the spawn and rpc *inside* the module. This is good practice because we will be able to change the internal details of the server without changing the client code. Finally, we get this:

Download area_server_final.erl

```
-module(area_server_final).
-export([start/0, area/2]).

start() -> spawn(fun loop/0).

area(Pid, What) ->
    rpc(Pid, What).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

To run this, we call the functions start/0 and area/2 (where before we called spawn and rpc). These are better names that more accurately describe what the server does:

```
1> Pid = area_server_final:start().
<0.36.0>
2> area_server_final:area(Pid, {rectangle, 10, 8}).
80
4> area_server_final:area(Pid, {circle, 4}).
50.2654
```

8.4 How Long Does It Take to Create a Process?

At this point, you might be worried about performance. After all, if we're creating hundreds or thousands of Erlang processes, we must be paying some kind of penalty. Let's find out how much.

To investigate this, we'll time how long it takes to spawn a large number of processes. Here's the program:

Download processes.erl

```
-module(processes).

-export([max/1]).

%% max(N)
%%   Create N processes then destroy them
%%   See how much time this takes

max(N) ->
    Max = erlang:system_info(process_limit),
    io:format("Maximum allowed processes:~p~n",[Max]),
    statistics(runtime),
    statistics(wall_clock),
    L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N,
    U2 = Time2 * 1000 / N,
    io:format("Process spawn time=~p (~p) microseconds~n",
              [U1, U2]).

wait() ->
    receive
        die -> void
    end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Here are the results I obtained on the computer I'm using to write this book, a 2.40GHz Intel Celeron with 512MB of memory running Ubuntu Linux:

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time=3.50000 (9.20000) microseconds
ok
2> processes:max(40000).
Maximum allowed processes:32768
=ERROR REPORT==== 26-Nov-2006::14:47:24 ===
Too many processes
...
```

Spawning 20,000 processes took an average of 3.5 μs/process of CPU time and 9.2 μs of elapsed (wall-clock) time.

Note that I used the BIF erlang:system_info(process_limit) to find the maximum allowed number of processes. Note that some of these are reserved, so your program cannot actually use this number. When we exceed the system limit, the system crashes with an error report (command 2).

The system limit is set to 32,767 processes; to exceed this limit, you have to start the Erlang emulator with the +P flag as follows:

```
$ erl +P 500000
1> processes:max(50000).
Maximum allowed processes:500000
Process spawn time=4.60000 (10.8200) microseconds
ok
2> processes:max(200000).
Maximum allowed processes:500000
Process spawn time=4.10000 (10.2150) microseconds
3> processes:max(300000).
Maximum allowed processes:500000
Process spawn time=4.13333 (73.6533) microseconds
```

In the previous example, I set the system limit to half a million processes. We can see that the process spawn time is essentially constant between 50,000 to 200,000 processes. At 300,000 processes, the CPU time per spawn process remains constant, but the elapsed time increases by a factor of seven. I can also hear my disk chattering away. This is sure sign that the system is paging and that I don't have enough physical memory to handle 300,000 processes.

8.5 Receive with a Timeout

Sometimes a receive statement might wait forever for a message that never comes. This could be for a number of reasons. For example, there might be a logical error in our program, or the process that was going to send us a message might have crashed before it sent the message.

To avoid this problem, we can add a timeout to the receive statement. This sets a maximum time that the process will wait to receive a message. The syntax is as follows:

```
receive
     Pattern1 [when Guard1] ->
         Expressions1;
     Pattern2 [when Guard2] ->
         Expressions2;
     ...
after Time ->
     Expressions
end
```

If no matching message has arrived within Time milliseconds of entering the receive expression, then the process will stop waiting for a message and evaluate Expressions.

Receive with Just a Timeout

You can write a **receive** consisting of only a timeout. Using this, we can define a function sleep(T), which suspends the current process for T milliseconds.

Download lib_misc.erl

```
sleep(T) ->
    receive
    after T ->
       true
    end.
```

Receive with Timeout Value of Zero

A timeout value of 0 causes the body of the timeout to occur immediately, but before this happens, the system tries to match any patterns in the mailbox. We can use this to define a function flush_buffer, which entirely empties all messages in the mailbox of a process:

Download lib_misc.erl

```
flush_buffer() ->
    receive
        _Any ->
            flush_buffer()
    after 0 ->
        true
    end.
```

Without the timeout clause, flush_buffer would suspend forever and not return when the mailbox was empty. We can also use a zero timeout to implement a form of "priority receive," as follows:

Download lib_misc.erl

```
priority_receive() ->
    receive
        {alarm, X} ->
            {alarm, X}
    after 0 ->
        receive
            Any ->
                Any
        end
    end.
```

If there is *not* a message matching {alarm, X} in the mailbox, then priority_receive will receive the first message in the mailbox. If there is no message at all, it will suspend in the innermost receive and return the first message it receives. If there is a message matching {alarm, X}, then this message will be returned immediately. Remember that the **after** section is checked only after pattern matching has been performed on all the entries in the mailbox.

Without the after 0 statement, the alarm message would not be matched first.

Note: Using large mailboxes with priority receive is rather inefficient, so if you're going to use this technique, make sure your mailboxes are not too large.

receive with Timeout Value of Infinity

If the timeout value in a receive statement is the atom infinity, then the timeout will *never* trigger. This might be useful for programs where the timeout value is calculated outside the receive statement. Sometimes the calculation might want to return an actual timeout value, and other times it might want to have the receive wait forever.

Implementing a Timer

We can implement a simple timer using receive timeouts.

The function stimer:start(Time, Fun) will evaluate Fun (a function of zero arguments) after Time ms. It returns a handle (which is a PID), which can be used to cancel the timer if required.

Download stimer.erl

```
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).

cancel(Pid) -> Pid ! cancel.

timer(Time, Fun) ->
    receive
        cancel ->
            void
    after Time ->
            Fun()
    end.
```

We can test this as follows:

```
1> Pid = stimer:start(5000, fun() -> io:format("timer event~n") end).
<0.42.0>
timer event
```

Here I waited more than five seconds so that the timer would trigger. Now I'll start a timer and cancel it before the timer period has expired:

```
2> Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

8.6 Selective Receive

So far we have glossed over exactly how **send** and **receive** work. **send** does not actually send a message to a process. Instead, **send** sends a message to the mailbox of the process, and **receive** tries to remove a message from the mailbox.

Each process in Erlang has an associated *mailbox*. When you send a message to the process, the message is put into the mailbox. The only time the mailbox is examined is when your program evaluates a **receive** statement:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard1] ->
        Expressions1;
     ...
after
    Time ->
        ExpressionTimeout
end
```

receive works as follows:

1. When we enter a **receive** statement, we start a timer (but only if an **after** section is present in the expression).
2. Take the first message in the mailbox and try to match it against Pattern1, Pattern2, and so on. If the match succeeds, the message is removed from the mailbox, and the expressions following the pattern are evaluated.
3. If none of the patterns in the **receive** statement matches the first message in the mailbox, then the first message is removed from the mailbox and put into a "save queue." The second message

in the mailbox is then tried. This procedure is repeated until a matching message is found or until all the messages in the mailbox have been examined.

4. If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled for execution the next time a new message is put in the mailbox. Note that when a new message arrives, the messages in the save queue are not rematched; only the new message is matched.
5. As soon as a message has been matched, then all messages that have been put into the save queue are reentered into the mailbox in the order in which they arrived at the process. If a timer was set, it is cleared.
6. If the timer elapses when we are waiting for a message, then evaluate the expressions ExpressionsTimeout and put any saved messages back into the mailbox in the order in which they arrived at the process.

8.7 Registered Processes

If we want to send a message to a process, then we need to know its PID. This is often inconvenient since the PID has to be sent to all processes in the system that want to communicate with this process. On the other hand, it is very *secure*; if you don't reveal the PID of a process, other processes cannot interact with it in any way.

Erlang has a method for *publishing* a process identifier so that any process in the system can communicate with this process. Such a process is called a *registered process*. There are four BIFs for managing registered processes:

register(AnAtom, Pid)
: Register the process Pid with the name AnAtom. The registration fails if AnAtom has already been used to register a process.

unregister(AnAtom)
: Remove any registrations associated with AnAtom.

  Note: If a registered process dies it will be automatically unregistered.

whereis(AnAtom) -> Pid | undefined
: Find out whether AnAtom is registered. Return the process identifier Pid, or return the atom undefined if no process is associated with AnAtom.

registered() -> [AnAtom::atom()]

Return a list of all registered processes in the system.

Using register, we can revise the example in Section 8.2, *A Simple Example*, on page 145, and we can try to register the name of the process that we created:

```
1> Pid = spawn(fun area_server0:loop/0).
<0.51.0>
2> register(area, Pid).
true
```

Once the name has been registered, we can send it a message like this:

```
3> area ! {rectangle, 4, 5}.
Area of rectangle is 20
{rectangle,4,5}
```

A Clock

We can use register to make a registered process that represents a clock:

Download clock.erl

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
    receive
        stop ->
            void
    after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

The clock will happily tick away until you stop it:

```
3> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1164,553538,392266}
TICK {1164,553543,393084}
TICK {1164,553548,394083}
TICK {1164,553553,395064}
4> clock:stop().
stop
```

8.8 How Do We Write a Concurrent Program?

When I write a concurrent program, I almost always start with something like this:

Download ctemplate.erl

```
-module(ctemplate).
-compile(export_all).

start() ->
    spawn(fun() -> loop([]) end).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop(X) ->
    receive
        Any ->
            io:format("Received:~p~n",[Any]),
            loop(X)
    end.
```

The receive loop is just any empty loop that receives and prints any message that I send to it. As I develop the program, I'll start sending messages to the processes. Because I start with no patterns in the receive loop that match these messages, I'll get a printout from the code at the bottom of the receive statement. When this happens, I add a matching pattern to the receive loop and rerun the program. This technique largely determines the order in which I write the program: I start with a small program and slowly grow it, testing it as I go along.

8.9 A Word About Tail Recursion

Take a look at the receive loop in the area server that we wrote earlier:

Download area_server_final.erl

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
```

```
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

If you look carefully, you'll see that every time we receive a message, we process the message and then immediately call loop() again. Such a procedure is called *tail-recursive*. A tail-recursive function can be compiled so that the last function call in a sequence of statements can be replaced by a simple jump to the start of the function being called. This means that a tail-recursive function can loop forever without consuming stack space.

Suppose we wrote the following (incorrect) code:

```
Line 1  loop() ->
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop(),
    5       someOtherFunc();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        ...
   10 end
```

In line 4, we call loop(), but the compiler must reason that "after I've called loop(), I have to return to here, since I have to call someOtherFunc() in line 5." So, it pushes the address of someOtherFunc onto the stack and jumps to the start of loop. The problem with this is that loop() never returns; instead, it just loops forever. So, each time we pass line 4, another return address gets pushed onto the control stack, and eventually the system runs out of space.

Avoiding this is easy; if you write a function F that never returns (such as loop()), make sure that you never call anything *after* calling F, and don't use F in a list or tuple constructor.

8.10 Spawning with MFAs

Most programs we write use spawn(Fun) to create a new process. This is fine provided we don't want to dynamically upgrade our code. Sometimes we want to write code that can be upgraded as we run it. If we want to make sure that our code can be dynamically upgraded, then we have to use a different form of spawn.

spawn(Mod, FuncName, Args)

This creates a new process. Args is a list of arguments of the form [Arg1, Args2, ..., ArgN]. The newly created process starts evaluating Mod:FuncName(Arg1, Arg2, ..., ArgN).

Spawning a function with an explicit module, function name, and argument list (called an MFA) is the proper way to ensure that our running processes will be correctly updated with new versions of the module code if it is compiled while it is being used. The dynamic code upgrade mechanism does not work with spawned funs. It works only with explicitly named MFAs. For more details, read Section E.4, *Dynamic Code Loading*, on page 437.

8.11 Problems

1. Write a function start(AnAtom, Fun) to register AnAtom as spawn(Fun). Make sure your program works correctly in the case when two parallel processes simultaneously evaluate start/2. In this case, you must guarantee that one of these processes succeeds and the other fails.

2. Write a ring benchmark. Create N processes in a ring. Send a message round the ring M times so that a total of N * M messages get sent. Time how long this takes for different values of N and M.

   Write a similar program in some other programming language you are familiar with. Compare the results. Write a blog, and publish the results on the Internet!

That's it—you can now write concurrent programs!

Next we'll look at error recovery and see how we can write fault-tolerant concurrent programs using three more concepts: links, signals, and trapping process exits. That's in the next chapter.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Erlang
http://pragmaticprogrammer.com/titles/jaerlang
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragmaticprogrammer.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/jaerlang.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com