erlang-in-lisp manual

August 18, 2008

Contents

1 User's Guide

Because Erlang-in-Lisp aims to copy the semantics of Erlang, much of this section highlights syntactic issues or places in which Erlang-in-Lisp differs from Erlang. It may be best to consult the Erlang documentation or an Erlang book if you are unfamiliar with its shared nothing approach to concurrency.

1.1 Toplevel

One of the advantages of Lisp and Erlang is the REPL. It allows for an interactive style of development. To have the full power of Erlang-in-Lisp at the REPL, too, it is necessary to call the `toplevel` function. After this function has been called, it is possible to send and receive messages at the REPL; it is a full-fledged Erlang-in-Lisp process.

The `toplevel` function also takes the concurrency strategy as a parameter. To call the `toplevel` function and use the unix process based concurrency (it is the default):

```
(toplevel)
```

or to specify the concurrency method explicitly:

```
(toplevel :process-class 'unix-process)
```

Likewise, to specify thread based concurrency:

```
(toplevel :process-class 'thread-process)
```

A second function, `toplevel-exit` can be called when Erlang-in-Lisp functionality is no longer desired at the REPL:

```
(toplevel-exit)
```

1.2 Spawning a Process

To spawn a process, simply pass the `spawn` function a name for your new process, and a thunk that is to be executed inside this new process. For example, to create a simple hello world process:

```
(spawn 'hello-world #'(lambda () (format t "hello world~%")))
```

`spawn` returns a process identifier, or pid, that is used for sending messages to the process.

1.3 Sending and Receiving Messages

1.3.1 send

The `send` function is used to send a message to a process using its process id. With the pid you simple:

```
(send pid 'this 'is 'my 'message)
```

Anything following the pid is sent as the message.

1.3.2 receive

Receiving messages from another process is simply a matter of calling the `receive` function with a block of code that contains the message patterns to match, and the code to execute when such a match is found. The receive pattern matching syntax follows that of fare-matcher. `receive` is a blocking call, and will not continue until a match has been found.

As an example, say we'd like to receive a message starting with the symbol `hello` followed by one variable (which we'll bind to `a`), and then print out that variable:

```
(receive
    ((list :hello a)
     (format t "The value of a is: ~A~%" a)))
```

1.3.3 receive-after

The `receive-after` function takes three parameters: a receive block, an integer timeout, and an after block. The recive block is the same as in the `receive` function, however, since `receive-after` is implemented with a destructuring bind, it is necessary to wrap the receive and after blocks in an extra set of parenthteses. The `receive-after` function blocks for at least the number of seconds specified as the timeout. If a message is still not found, the code in the after block is executed.

Now we extend our receive example above to print a timeout message after 5 seconds if a message has not been received:

```
(receive-after
    (((list :hello a)
     (format t "The value of a is: ~A~%" a)))
    5
    ((format t "No message received.~%")))
```

This syntax is certainly a bit tedious and is on the todo list of things to change/improve.

1.4 Errors and Linking Processes

The Erlang-in-Lisp error system uses the condition system of Common Lisp directly. All errors that are not handled inside of a process (and thus interrupt the normal flow of that process) are converted to messages and sent to linked processes. At the moment, this means that all processes behave like Erlang system processes, and therefore there is no 'cascading failure' behavior.

1.5 A Simple Example

Below is the simple ping-pong example that is often to introduce the the ideas of Erlang. It combines some of the functions discussed above. The `ping` function takes a pid and an integer, `count` as arguments. It then sends `count` messages to the pid, waiting after each message for a response.

The pong function simple accepts ping messages and responds to each with a pong until the receipt of a done message, at which point the process exits.

```
(defun ping (pong count)
  (loop for iter from 1 to count
     do (send pong :ping iter (self))
     do (receive
     ((list :pong) (format t "PING -- Got pong.~%"))))
  (send pong :done))



(defun pong ()
  (receive
    ((list :ping iter ping)
     (format t "PONG -- Got ping.~%")
     (send ping :pong)
     (format t "PONG -- Sent pong.~%")
     (pong)) ;;check that this tail call can be eliminated.
    ((list :done)
     (format t "PONG -- Got done."))))

(defun ping-pong (count)
  (let* ((pong (spawn 'pong #'pong)))
(ping (spawn 'ping #'ping pong count))))
```

2 Hacking on erlang-in-lisp

This section is for those interested in in helping with the development of erlang-in-lisp.

2.1 Getting the Sources

erlang-in-lisp is kept in git repository. To get the code with git, create a new repository and pull in the lastest version:

```
mkdir erlang-in-lisp
git init
git pull http://common-lisp.net/project/erlang-in-lisp/git/ master
```

To get the sources compiled and running, you will need to the appropriate dependencies (see section 2.2).

Obviously there are the regular git facilities to create branches, etc, but once changes are ready for the mainline you can push a patch to the master branch (the assumption is you are in the erlang-in-lisp group on common-lisp.net):

```
git push USER@shell.common-lisp.net/project/erlang-in-lisp/git
```

2.2 Dependencies

All the dependencies of erlang-in-lisp are either stored directly in the repository or can be fetched with the makefile in the `deps` directoy. Simple change to this directory and issue the `make get-deps` command. The dependencies will download (it can take a while). For this process to complete normally, your system will need to have darcs, wget, tar, gzip, and git installed. You can remove the downloaded dependencies with `make clean`.

Note that this makefile is not very robust and should probably be replaced with something more maintainable.

2.3 Directory Layout

Though most of these are self-explanatory, a brief description of the directory structure as it currently stands follows:

- `src` – erlang-in-lisp sources are stored here, including the asdf system definition file.
- `test` – Tests are stored here. This includes a separate system defintion for testing purposes (as it may have its own dependencies, etc). See section 2.2.
- `deps` – Dependencies for erlang-in-lisp are stored here. Some are directly in the repository, and the other dependencies can be fetched with the included makefile.
- `website` – Files for the website on common-lisp.net are stored here.
- `doc` – Documentation, including this manual, is stored here.

2.4 Running Some Examples

To get started, you must first load the `erlang-in-lisp` package. In SBCL, this is as simple as `(require 'erlang-in-lisp)` when you are in the same directory as `erlang-in-lisp.asd`. Once that is done, it is simple to run some examples. Below, we run the ping-pong example:

```
CL-USER> (in-package #:erlang-in-lisp)
#<PACKAGE "ERLANG-IN-LISP">
ERLANG-IN-LISP> (toplevel :process-class 'thread-process)
#<THREAD-PROCESS {A917FD1}>
ERLANG-IN-LISP> (ping-pong 3)
pong started
ping started
PONG -- Got ping.
PONG -- Sent pong.
pong started
PING -- Got pong.
PONG -- Got ping.
PONG -- Sent pong.
pong started
PING -- Got pong.
PONG -- Got ping.
PONG -- Sent pong.
pong started
PING -- Got pong.
PONG -- Got done.
1
ERLANG-IN-LISP> (toplevel-exit)
```

Because the concurrency strategies of erlang-in-lisp are interchangeable, the only change to run the above example with unix-process based concurrency is:

```
ERLANG-IN-LISP> (toplevel :process-class 'unix-process)
```

Likewise if any other concurrency strategies are defined in the future, the name of the class that holds the process metadata for that strategy need only be specified.

2.5 Unit Tests

Erlang-in-Lisp uses the fiveam testing framework. For now the tests are embedded direcly in the files alongside the code, but this may become untenable if extra dependencies are introduced (though we could use the `#-fiveam` read macro as well).

2.5.1 Running the Tests

To run all the tests simple type:

```
ERLANG-IN-LISP> (run!)
```

To run on specific test (here, for example, the `simple-send-receive-test`):

```
ERLANG-IN-LISP> (run 'simple-send-receive-test)
```

2.5.2 Creating new Tests

Tests are found at the end of the `.lisp` file whose code they test. See `eil.lisp` as en example.

Here is a simple unit tests that sees whether or not five is equal to five:

```
(fiveam:in-suite erlang-in-lisp::eil-suite) ;;included once before unit tests
(fiveam:test five-equality
  "Test whether five is equal to five."
  (is (eql 5 5)))
```

3 Design

Erlang-in-Lisp has a design inspired by the metaobject protocol. There is a thin, user-facing macro layer that in turn calls an extensible generic function based layer. The macro layer can be found in `core-eil.lisp`; higher level abstractions built on top of this layer can be found in `eil.lisp`. The heart of the generic function layer is in `process.lisp`. Two extensions of the generic function layer that provide the concurrency strategies discussed below can be found in `process-unix.lisp` and `process-threads.lisp`.

3.1 Fork Based Concurrency

The original plan for Erlang-in-Lisp was to start with fork as the primary method for spawning language-level processes using a "one OS process per Erlang process" model indeed. The reasons follow.

1. Yes, the performance of fork doesn't scale and thus severely limits the applicability of the actor paradigm, but scaling can be achieved later by other means: portable lisp-in-lisp green threads (as in the Tube or philip-jose), or implementation-specific green threads.

2. fork is portable (to any unix/cygwin), and provides desirable isolation properties that would require a large number of hours of debugging to enforce using other methods (especially when Lisp implementations and libraries were not designed to be thread-safe). Yes, these other methods will have to be implemented eventually, but fork-based concurrency will allow us to have a reference implementation by then that can help discriminate between implementation bugs vs application bugs.

3. Fork-based concurrency also provides for very desirable isolation from bugs in third-party libraries, and for native compilation of lisp code without an intermediate translation layer - two properties that we will certainly want to preserve even when we have other implementation strategies available for multiprogramming.

4. last but not least, fork-based concurrency seems like it's the easiest, cheapest and fastest way to get an initial working system.

So no, fork-based concurrency not the be-all end-all of Erlang-in-Lisp, but I really think it's the best way to start.

3.2 Thread Based Concurrency

Some simple thread based concurrency has been implemented on top of the bordeaux-threads library. Because the threads share the same namespace, there is plenty of room for programmer mischief. However, if no messages are mutated, all is well and the system should work properly.

It is implemented as a simple (and likely inefficient) lock and condition-variable based system. Currently its main goal is to show that pluggable concurrency strategies are possible and to organically evolve the mechanism by which the various concurrency strategies are implemented.

4 Todo

4.1 Receive and Timeouts

First off, it would be nice to eliminate `receive-after` and instead have one function, `receive` with some `loop`-like syntax where we could include a timeout and a block after an `after` statement. Perhaps this is not very Lispy, but neither is `loop`.

Also, timeouts are not currently not implemented in the thread based concurrency (we need to somehow parameterize our wait on the condition variable) and may be poorly defined in the unix based concurrency. In the unix based concurrency, the timeout is interpretted to mean how long the process blocks for a message *after* it has already searched the mailbox for matches. While the timeouts in Erlang are certainly meant to be soft, this particular implementatation could lead to problems if the mailbox is very full. A more sophisticated timing mechanism is needed.

4.2 Process Linking

Right now all unhandled Lisp conditions are convered to messages and sent to linked processes. More testing of this is needed.

Also, all processes are Erlang system processes. Thus, processes that are linked but not system processes should be made to fail when necessary. This could be tricky, but may not be a huge priority.

4.3 Serialization

Currently the pids are serialized with read and print (this is necessary in order to send the pids as messages). It is a huge kludge. There must be a nicer way to do this.

One of the nicer ways is using cl-store, and some attempts have been made in the direction (the attempts remain in `process-unix.lisp` and are commented out). A good discussion of all of the options is available at: http://www.pentaside.org/paper/persistence-lemmens.txt

Pids are not the only problem; to acheive true distribution, we must be able to serialize closures and send them over the wire. This has been accomplished in Termite with Gambit-C, but we would also like it to be portable across Lisp implementations. Some possible solutions may be found in the green threads discussion below.

4.4 Green Threads

It would be nice to implement a concurrency strategy that stay's true to Erlang's lightweight green threads. Several approaches:

- a portable but very slow meta-programming green threads as in philip-jose (uses arnesi's slow ((Lisp with continuations) in Lisp) interpreter), that should allow migration of processes (within some restrictions).
- a similar portable but less slow method using a lisp-in-lisp compiler instead like the one in Screamer, which might not allow for migration (or might, if we use the same tricks as the Tube).
- pushing the code to a remote Erlang implementation (whether Erlang-in-Lisp or plain Erlang), if it doesn't use any of our Lisp-specific ”extensions”, but only things available with say LFE.

4.5 Erlang Interface

It would be nice to have Erlang-in-Lisp programs that are able to interact with existing Eralng nodes and processes. One way to do this would be to create bindings to the erl_interface C library. This is probably most easily done with the cffi groveler. Lispier bindings could be created later.

4.6 Parsing Erlang

An utlimate goal is to arse Erlang with something like ometa. Ideally we would not parse Erlang directly, but instead parse core erlang, which is specified in this document. Parsing Core Erlang should be much simpler and allows us to use the existing Erlang front end to deal with any language changes that are not reflected in the more stable Core Erlang spec.

4.7 Process Monitoring

Watching other processes is hard to do reliably in Unix. If the processes are linked by parent/child relationships, then the parent can easily watch if children die with SIGCHLD (and *must* do it to avoid zombies) - but this is not general enough mechanism for Erlang-in-Lisp because we are mapping a graph of Erlang processes onto the unix process tree.

Another option is to share a pipe with the other process before you even fork, and add the watching of it in your epoll loop, so you can say the process is dead when you get an EPOLLHUP. This depends on all the considered processes following the proper convention of sending around the fd to the pipe, and so is not applicable to watching arbitrary other processes, but may work.

It might be possible to open an fd on some file in /proc/$pid and receive an event when the process dies.

These solutions require experimentation, but have the advantage that they are not subject to the same race conditions as having to register with a central server after you've forked.