COSC441

Lecture 8
Introduction to Erlang

Approach

- The reference book is “Learn You Some Erlang for Great Good!” by Fred Hébert.
- http://learnyousomeerlang.com lets you read it free on-line.
- What I am going to do is to introduce you to some of the key ideas; turn to the book for details.
- https://www.erlang.org/docs reference material

What goes into a language?

- Common ideas
- Application-specific ideas
- Designer quirks
- Accidents of history

Whence Erlang?

- Erlang was invented **by** Joe Armstrong, who was familiar with a wide range of imperative, OO, and declarative languages, specifically including Lisp, Prolog, and Strand-88.
- It was invented **at** the Ericsson Computer Science Laboratory.
- It was invented **for** programming soft real-time distributed telecoms applications, and secondarily for Internet applications.

Common ideas

- What do we need for a programming language?
- Minimally, a few combinators.
  - I x = x
  - K x y = x
  - S x y z = x z (y z)
  - https://en.wikipedia.org/wiki/SKI_combinator_calculus
- What about numbers, sequences, &c?
  We can model those as functions!

Common ideas 2

- We need some built-in data types.
- We need operations for constructing them, deconstructing them, deriving new ones, and so on.
- We need a way to give names to values.
- We need a way to define and name functions.
- We need some form of conditional construct.

Built-in data types

- Integer (unbounded)
- Float (IEEE 754 doubles)
- Atoms (uniquely stored strings)
- Lists [] (empty) [Head|Tail] (non-empty)
- Tuples {X1,...,Xn}
- Maps #{Key=>Val, …, Key=>Val}
- Functions as values

Application data types

- Binaries (originally byte sequences, now bit sequences) for shipping uninterpreted packets around.
- Process IDs
- References (unforgeable unique “cookies”)
- Ports (I/O connections that look a lot like pids)

Variables

- In the shared memory model, we saw that shared mutable data can be trouble.
- Let's not have any mutable variables!
- (There are actually several mutable bulk stores: the file system, the module system, the process registry, and several kinds of key-value store.)
- X = 1 is **not** an assignment statement!

Variables 2

- A “don't care” variable or “wild card” is written as a single underscore _
- Named variables are identifiers beginning with a capital letter.
- A variable is not a box you can change, but an alias for a value. A variable can become bound once and only once.
- See also “Single Assignment C”.

Matching 1

- Variable = Expression
  - evaluates Expression to V
  - if Variable has no value yet, binds it to V
  - if Variable has a value, checks whether that value is equal to V.
    - If it is not, raises an exception.
  - The whole form has V as its value, so (X = 1) + 1 has the value 2.

Static or dynamic typing?

- One purpose of type checking is to ensure that the whole program is consistent.
- One of the core requirements for Erlang was to support “hot loading” where a module can be replaced while the program is running. It is common for Java to load classes at run time, but Java does not **replace** classes dynamically
- Since Erlang doesn't *have* the whole program, it was originally designed without a type system.

Recognisers

- If you don't have types, you need recognisers.
- is_integer/1, is_float/1, is_number/1, is_atom/1, is_list/1, is_tuple/1, is_function/1, is_function/2, is_reference/1, is_pid/1, is_binary/1.
- We refer to the function called *name* with *arity* arguments in *module* as *module:name/arity* or *name/arity* for short.
- A recogniser returns the atom true or false.

Common operations

- The usual arithmetic operations
- Comparison. number < atom < reference < fun < port < pid < tuple < list < binary.
- Two sets of equality operators: =:= and =/= require type match as well as value match, while == and /= will equate integers and floats.
- Trap: equal-to-or-less-than is spelled =< . Erlang got this from Strand-88 which got it from Prolog which got it from Pop-2 (1967).

Conversions

- Several built-in operations convert a value from a *source* type to a *target* type. The convention is that such operations are called *source*_to_*target*/1.
- “foo” is a string. A string is just a list whose elements are Unicode code-points.
- <<”foo”>> is a binary. This will be a sequence of 3 bytes.
- list_to_binary(“foo”) => <<”foo”>>
- binary_to_list(<<”foo”>>) => “foo”

Functions

- fun (Arguments) -> Body end
  is an anonymous function.
- myfun(Arguments) ->
  Body**.**
  is a named function. Notice the full stop at the end. Every Erlang top level form must end with a full stop.
- Arguments are passed by pattern matching.

Creating a new process

- spawn(*Function_With_No_Arguments*) creates a new process executing the given function and returns its process ID.
- To find your own process ID, use self().
- The new process does not know anything about its parent unless you tell it. There is no equivalent of getppid().
-

Communication

- *Pid* ! *Message*
  sends *Message* to the process named by *Pid.*
  - *Pid* can be a process id.
  - Or an atom referring to a process in the registry.
  - Send a message to a dead process and it just quietly disappears.
  - If *Pid* is not a pid or a registered name, that's an error.
  - *Message* can be any value, including a function.

Receiving a message

- receive
  Pat [when Guard] -> Body
  ; Pat [when Guard] -> Body
  ; …
  ; after Timeout -> Body
  end.