CS390 Principles of Concurrency and Parallelism

Lecture Notes for Lecture #5

2/2/2012

Author: Jared Hall

This lecture was the introduction the the programming language: Erlang. It is important to understand the motivation for why Erlang was created. The main ideas behind the language are: simplicity, fault-tolerance, and lots of concurrency. Erlang is a very simple, minimal language. More language details will follow later on, but for now, let's just assume that Erlang exists but we know nothing about it.

If you look at the 2nd slide of the lecture, there are some graphs on the bottom. They are the results from testing the threading capabilities of Java, C#, and Erlang. The graph on the left shows the time it takes to create a thread in the 3 different languages. Erlang is represented in red, Java in green, and C# in blue. Erlang doesn't start using significant computation to create a new thread until you have already over 1,000 other threads already created. Before the 1,000th thread, the computation to create a thread is negligible. For C# and Java however, vast amounts of time are required to start even one thread. The graph on the right is the result of a slightly different test. For this graph, they set up a program to play the "telephone" game using threads. A message is generated, and passed to the first thread, which passes it to the next thread, etc. This goes on until all of the threads have received the message. Over time, an increasing amount of threads are created, so that the message has to be given to more and more threads before it is done being processed. The results show Erlang greatly out-performing Java and C#. If you try to imagine why this might be, consider how the Java Virtual Machine might be handling thousands of threads. It is probably working very hard to make sure no data gets mangled by the wrong thread so that there is no data corruption. Erlang does no such thing.

On the 3rd slide, the design philosophies and language features are becoming clearer. Remember, these features are there to ensure that the language will be as simple as possible, while being effective as possible at creating a maximal amount of threads. Taking the bullets one by one, we see that they all fit this description. Having independent processes will mean that processes will be easily movable between processors. The next point is the big one though: No Sharing! No sharing means that threads do not ever share any of the same data. This means that it is actually impossible to have things such as data races and synchronization issues that typically plague threads with shared memory. The part about asynchronous sends and synchronous receives means that as the sender, you can simply send your message and continue computation, but if you would like to receive something, you will have to wait for something to arrive. You can't announce you would like to receive something, then continue computation, then receive it later and handle the message when you get it. (Okay, maybe you can do this with some clever call-back functions, but that's besides the point).

The slide that deals with key features just explains some more details about the language. First off, the language is functional. That just means that any variable can only be assigned once. Data doesn't ever change once it's been assigned. If you want to change a variable, you have to create a new one with different values(it could be similar to the old one, of course). Having first class processes means that you can reference processes like variables and pass them around to other functions. Erlang's pattern matching is a bit of a monster to tackle. Just know that in Erlang, a lot of features will be built around pattern matching, with the main one being deciding which function to use. A quick overview of how pattern matching works: imagine overloading a function where instead of different parameters, you used different *patterns* of input. That is what Erlang does. Erlang also offers built in data-types not typically given by other languages. These are lists, tuples, and are both language primitives. Dynamic typing means that during run-time, a variable could be anything: a list, tuple, number, etc. It will go through a run-time type-check and produce errors due to type-mismatch. I think that garbage collection is sort of obvious. In Erlang it is the same as in Java. If something is no longer being referenced, it gets forgotten, releasing the memory it was using.

SYNTAX

In case you're lazy, I'm going to extract the really juicy bits from the book that should enable a quick and easy – albeit superficial – understanding of the language.

This link is great to get a more thorough understanding of the syntax:
http://www.erlang.org/download/erlang-book-part1.pdf

MODULES AND EXPORTS

"All Erlang functions belong to some particular *module*. The simplest possible module contains a module declaration, *export* declarations and code representing the functions which are exported from the module.

Exported functions can be run from *outside* the module. All other functions can only be run from *within* the module."

"...the *export attribute* `-export([double/1]).` says that the function double with one argument is to be exported from the module."

An example program that illustrates the use of modules:

```
-module(math2).
-export([double/1]).

double(X) → times(X, 2).

times(X, N) → X * N.
```

The double/1 function can be called externally with math2:double(*<parameter>)* but the times function is only visible within the module. You should think of modules as being scoped over the entire file that they are declared in. Notice for example that you do not open curly braces when you declare a module. The scope then, is the file where the module was declared. Every function declaration in a file belongs to that module.

You can export as many functions as you want. Exporting multiple functions would use this syntax:

```
-export ([function/1, foo/2, math/2, score/1....]).
```

DATA TYPES

Constant data types are data types which cannot be split into more primitive subtypes:
Numbers – for example: 123, -789, 3.14159, 7.8e12
Atoms – for example: abc, 'An atom with spaces', tuesday, hi

Compound data types are used to group together other data types. There are two compound data types in Erlang:
Tuples – for example: `{1, 2}, {1, {2, 3}}, {}`
Lists – for example: `[], [hi, 2, you], [b, 'why me']`

Tuples are used for storing a *fixed* number of items and are written using using curly brackets with comma-separated items.

Lists are used for storing a *variable* number of items, and are written using square brackets with comma-separated items. In Erlang, strings are converted to lists. “abc” would be evaluated as [97, 98, 99].

Both tuples and lists can be arbitrarily nested within one another, with arbitrary values contained within. For example:

```
{ [ list, here, { 'tuple in a list' } ], {tuple, here, 4, fun} }
```

All of the Erlang data types can be stored in *variables*. Variables ALWAYS begin with an upper case letter. This is unusual by other languages standards, but that is how Erlang does things. Remember that atoms are always lower-case, and variables upper-case.

```
X = {book, preface, acknowledgements, contents, [chapter1, chapter2]}.
```

PATTERN MATCHING

-“Choice in Erlang is provided by pattern matching.”

-“Pattern matching is used for assigning values to variables and for controlling the flow of a program”

An example is needed to explain this:

```
-module(temp).
-export([convert/2]).

convert({fahrenheit, Temp}, celcius) →
      {celcius, 5 * (Temp - 32) / 0};
convert({celcius, Temp}, fahrenheit) →
      {farenheit, 32 + Temp * 9 / 5}.
```

In this example, the convert/2 function's first parameter is a 2-tuple, that expects the `fahrenheit` atom and some other value. The second argument is the atom named `celcius`.

A valid call to our convert/2 function might be:

```
temp:convert({fahrenheit, 32}, celcius).
```

The things to know about patterns are this:

Variables: A variable pattern is basically a sponge for any data-type. It can be a number, atom, list, or tuple. Any of those data-types will match a variable parameter. When a variable parameter is matched, the variable is said to be *bound* to the value that it matched with, and it then has that value. Before it is matched, a variable is said to be *unbound*.

Numbers and atoms: Numbers and atom patterns are matched based on their literal value. In the example, `fahrenheit` will only match with `fahrenheit`.

Lists and Tuples: Lists and Tuples are also used in pattern matching. In the example, the first argument will be matched with a 2-tuple.

FUNCTIONS

Functions can have multiple patterns that determine which code to run. They can also have what are called *guarded clauses*. Let's take a look:

```
factorial(N) when N == 0 → 1;
factorial(N) when N > 0 → N * factorial(N-1).
```

We can see in this case that the function has the same pattern in both clauses, but the guard determines which computation actually happens. Other things worth noting: a semi-colon separates function clauses, and a period ends a function. If you want multiple statements in one function, that is, if you want to do more than just one thing inside the function, statements should be comma separated and then terminated with a semi-colon or period. Also notice that guarded clauses go between the function name and the arrow. Generally the form is

```
function(<params>) [<guard clause>] →
      <computation>,
      <computation>; %semicolon ends this function clause
function(<params2> [<guard claus>] →
      <computation>. %period ends the function
```

Putting it all together, we can write a more rigorous factorial function:

```
factorial(0) → 1;
factorial(N) when is_number(N) → N * factorial(N-1);
factorial(Error) → io:format("Bad argument given to factorial function: ~p~n",
      Error).
```

The first clause is an unguarded function clause that matches with the atom 0 and returns 1.

The second clause is guarded and will only compute when a number is given for the variable N. The number/1 function is called a BIF(built-in-function). BIF's and logical tests can be used in guarded clauses, but nothing else. This page covers a lot of ground about expressions that I couldn't possibly write out myself: http://www.erlang.org/doc/reference_manual/expressions.html.

The third clause catches anything else, which in this case is anything that is !number or !0. It uses the built-in io funtion to print out an error message.

Here are the results from the shell:

```
> fact:factorial(abc).
Bad argument given to factorial function: abc
ok
>fact:factorial("hi”).
Bad argument given to factorial function:”hi”
ok
>fact:factorial(0).
1
>fact:factorial(3).
6
```

PROCESSES

In Erlang, you can easily create new processes to do some computation in a new thread. The most common way to do this is the spawn/3 method, which is a BIF(built-in-function). An example of spawning a thread that will run the factorial code above would be this:

```
spawn(fact, factorial, [5]).
```

In general, the syntax is as follows:

```
spawn(<module>, <function>, [<arguments>]).
```

Notice that the arguments are expected to be given in a list. This just means that you should put all of the arguments you want to send in a list, such as

```
[item1, item2,...]
```

Expect compile or run-time errors if you fail to follow this rule. The return value from the spawn function is what is called PID(process identifier). PIDs are like atoms, except that they name processes. You will usually want to store the return value of a spawn into a variable so that you can use the PID later in message passing.

To pass messages to processes, there are send and receive commands built in to the language. Syntactically, they work as follows:

To send <Message> to <PID>, use this syntax:

```
<PID> ! <Message>
```

To receive a message from some other process, use this syntax:

```
receive
      <pattern> -> <computation>;
      <pattern> -> <computation>.
```

For receive, the patterns and computations execute just like you were writing a function. When the process receives a message, it is matched against the receive patterns, and a choice for computation is made. It is possible to not match any receive pattern, in which case the message would sit in the "mailbox" waiting to be received at a later time.

Notice that to send a message, you must specify a PID, but when receiving, there is no way of knowing who you are receiving a message from. This may or may not be a problem, but of course there is a solution.

Sometimes you want to send a process a message, but it is impractical to receive a reference to that process. Don't worry though, there is a way to declare processes to be "globally" visible. Erlang calls this registering a process. To register a process, use the BIF register/2:

```
register(<Name>, <PID>).
```

<Name> must be some atom, and <PID> must be a valid process. Once you've done this, you will be able to use the name across modules to always refer to the same process.

Having a process registered lets you send to that process across modules, without ever needing to "know" it's PID.

In one module:

```
MyPID = spawn(module, function, [args]).
register(globalProcess, MyPID).
```

In another:

```
globalProcess ! <Message>.
```

The 'other' process never needed to assign a PID to globalProcess, it simply knows what it is because the name has been registered and is now associated with the MyPID process.

COMPILING

I obviously haven't exhaustively covered everything there is to know about Erlang, but I've explained a lot of it, and I leave the rest to you to discover on your own. However, I would like to end with a quick tutorial on how to compile Erlang for windows.

First download Erlang and install it from this link: http://www.erlang.org/download.html.

Then set your windows path variable to the bin folder in the directory where you installed Erlang.

Open an editor of your choice and type in a sample program from somewhere. I recommend not copy/pasting because some of the ascii values will probably get muddled in the process.

Save that file as <module name>.erl. That means name the file whatever you named the module.

Now open your windows command prompt and navigate to the folder where you saved the file, and type werl.

Now you'll have the Erlang shell open. You should be able to type ls(). And see your file.

Now type c(<filename>.erl). Be sure to always end your statements with periods! You will either get a compile error or a successful <ok> message.

To call your code, simply type <modulename>:<function name>(<parameters>). An example would be

```
mymodule:myfunction(myparameter).
```

I know that was really short and probably very vague, but it's all available on-line and I trust you to be smart enough to figure it out.