Introduction to Erlang

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – January 5, 2018

Objectives

At the end of this lecture, you should be able to:

- Explain key concepts of functional programming:
  - What is referential transparency?
  - Why do functional languages use recursion instead of loops?
- Read and write simple Erlang functions
  - Write functions using explicit recursion and using `fold` for the sum, product, maximum, or the longest run of ones in a list
  - How does the `fold` operator encapsulate a common programming pattern?

A.K. Erlang
1878 – 1929
Danish mathematician, engineer, and statistician

picture from https://upload.wikimedia.org/wikipedia/commons/f/fd/Erlang.jpg

Erlang

- Message passing is **our first parallel programming paradigm:**
  - We'll use Erlang as a message-passing language.
  - Many concepts, algorithms, and performance trade-offs for parallel programming are easy to illustrate using Erlang.
  - Erlang has features that make common parallel programming pitfalls easier to avoid.
- Erlang is **functional:**
  - Values are bound to variables when the variable is declared.
  - The value of a variable **never** changes.
  - For parallel programming, this means you don't have one process corrupting the state of another process.
- Erlang uses **message passing** for inter-process communication:
  - Communication between processes is with send and receive expressions.
  - Communication costs are the dominant design issue for most parallel software
    - ⋆ Erlang makes these issues explicit.

Functional Programming

- Imperative programming (C, Java, Python, . . .) is a programming model that corresponds to the von Neumann computer:
  - A program is a sequence of statements.
    In other words, a program is a recipe that gives a step-by-step description of what to do to produce the desired result.
  - Typically, the operations of imperative languages correspond to common machine instructions.
  - Control-flow (`if`, `for`, `while`, function calls, etc.)
    Each control-flow construct can be implemented using branch, jump, and call instructions.
  - This correspondence between program operations and machine instructions simplifies implementing a good compiler.
- Functional programming (Erlang, lisp, scheme, Haskell, ML, . . .) is a programming model that corresponds to mathematical definitions.
  - A program is a collection of definitions.
  - These include definitions of expressions.
  - Expressions can be evaluated to produce results.
- See also: the *LYSE* explanation.

Declarative Programming – another way to say "Functional"

- A program is a collection of definitions, also called **declarations**.
- Examples:

```
count3s([]) -> 0;
count3s([3 | Tl]) ->
  1 + count3s(Tl);
count3s([_ | Tl]) ->
  count3s(Tl).
```

```
factorial(0) -> 1;
factorial(N)
    when is_integer(N), N > 0 ->
  N*factorial(N-1).
```

- Functions are defined with a set of declarations that are in the style of a mathematical definition.
  - The definitions that we can write are restricted by the syntax of the language so there is an "obvious" way to execute the program.
  - More formally, "obvious" means λ-calculus.

Referential Transparency

- This notion that a variable gets a value when it is declared and that the value of the variable never changes is called **referential transparency**.
  - You'll hear me use the term many times in class – I thought it would be a good idea to let you know what it means. ☺
- We say that the value of the variable is **bound** to the variable.
- Variables in functional programming are much like those in mathematical formulas:
  - If a variable appears multiple places in a mathematical formula, we assume that it has the same value everywhere.
  - This is the same in a functional program.
  - This is not the case in an imperative program. We can declare `x` on line 17; assign it a value on line 20; and assign it another value on line 42.
  - The value of `x` when executing line 21 is different than when executing line 43.

Loops violate referential transparency

```
// vector dot-product
sum = 0.0;
for(i = 0; i < a.length; i++)
  sum += a[i] * b[i];
```

```
// merge, as in merge-sort
while(a != null && b != null) {
  if(a.key <= b.key) {
    last->next = a;
    last = a;
    a = a->next;
    last->next = null;
  } else {
    ...
  }
}
```

- Loops rely on changing the values of variables.
- Functional programs use recursion instead.
- See also the *LYSE* explanation.

Sequential Erlang

- Example: from `count3s` to `sum`.
- Encapsulating patterns with higherorder functions.
- Tuples

In Class Coding: list sum

- Compute the sum of the elements of a list.
- Review: The count 3s example from the Introduction slide deck:

```
count3s([]) -> 0; % The empty list has 0 threes

% A list whose head is 3 has one more 3 than its tail
count3s([3 | Tail]) -> 1 + count3s(Tail);

% A list whose head is not 3 has the same number of 3s as its tail
count3s([_Other | Tail]) -> count3s(Tail).
```

- In-class exercise: Apply the same pattern to create `sum/1`.

In Class Coding: list sum

```
sum([]) -> _____;
sum([Hd | Tl]) -> _____.
```

- `count3s` had three patterns, but there are only two patterns in the template above.
- Do these two patterns cover all the cases?
- Do we need other patterns?
- Why?

The Pattern

- Based on the functions for `count3s` and `sum`, how would you write functions for:

```
prod(List) ->  the product of the elements of List.
max(List) ->   the largest element of List.
               % max(List) has a "tricky" detail, the function max/2 is
               % an Erlang built-in-function. We are defining max/1.
               % The compiler has no problem telling them apart.
```

- There's a common pattern to all of these:
  - Given an operation, `Op(X, Y)`, and a list, `List`,
  - combine all of the elements of `List` using `op`
  - If `List = [E1, E2, E3, ..., EN]`, we want to compute
  - `op(E1, op(E2, op(E3, ..., EN) ...))`

The Fold Function

- The `fold` operator:

```
% fold_v1(Op, List) -> combine the elements of List using Op
fold_v1(_Op, [E]) -> E;
fold_v1(Op, [Hd | Tl]) -> Op(Hd, fold_v1(Op, Tl)).
```

- Writing `sum` using `fold_v1`

```
sum_using_fold_v1(List) ->
    fold_v1(fun(E, Acc) -> E+Acc end,
            List).
```

  - But what if `List` is empty?
  - Same problem occurs for `prod`, `max`, `count3s`, etc.

- Solution: include an initial value for the accumulator `Acc` as a parameter to `fold`.
- In-class exercise: Implement `fold/3` with an initial value `Acc0` for the accumulator.

Fold in the Erlang Library

- The standard Erlang library provides two versions of `fold`
  - `foldl(Op, Acc0, List)` combines the elements of `List` from left-to-right.
  - `foldr(Op, Acc0, List)` combines the elements of `List` from right-to-left.
  - `foldl` should be used if `List` can be very long because `foldl` is **tail recursive** (see next lecture).

Higher-Order Functions

- `fold` is an example of a higher order function.
  - `fold` takes a function as an argument, `Op`.
  - `fold` uses this function to implement the specific fold operation that the programmer wants.
- Higher-order functions can
  - Take functions as arguments (e.g. `fold`), or
  - Return functions as a result.
- In-class exercise: Implement `prod/1` and `max/1` using a fold.

More Fold Examples

- Longest run of zeros.
  - `longest_0_run(List) ->` the length of the longest sequence of consecutive `0`s in `List`.
  - Example: `longest_0_run([0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1]) -> 4`.
  - Can we implement `longest_0_run(List)` using `fold`?
  - Similar problems:
    - ★ `longest_run(Key, List) ->` the length of the longest sequence of consecutive elements with value `Key` in `List`.
    - ★ `longest_ascending(List) ->` the length of the longest sequence of consecutive ascending values in `List`.
- What makes these problems different than our original examples for `fold` (i.e. `count3s`, `sum`, `prod`, and `max`)?

Accumulators with multiple values

- We need a way for our accumulator function to return multiple values.
- For the `longest_run(Key, List)` problem, what does the accumulator need to contain?
- If you find functional programming to be a new way of thinking:
  - Just for a moment, think of how you would write `longest_run` using a for-loop or while-loop?
  - What variables did you use to pass information from one loop iteration to the next?
  - This should let you know what you need to use for the accumulator value for an implementation using `fold`.
- Once we've figured out what should be in the accumulator, we could use
  - A list: the Dr. Racket approach.
  - A tuple: generally better Erlang "style"
  - Erlang also has records, but the syntax is kind of awkward. Even so, records can be the right choice when making bigger systems (i.e. bigger than what we'll be doing in class).

Tuples

- tuples:
- Construction: `{1, dog, "called Rover"}`
- Operations: `element`, `setelement`, `tuple_size`.
- Lists vs. Tuples:
  - **Lists** are typically used for an **arbitrary** number of elements of the same “type” – like arrays in C, Java, ...
  - **Tuples** are typically used for an **fixed** number of elements of the varying “types” – likes a `struct` in C or an object in Java.
  - Tuples often make good accumulators for `fold`.

`longest_run(Key, List)`

- Design questions:
  - What should the elements of the accumulator tuple be?
  - What should the initial value of the accumulator, `Acc0`, be?
  - How should a new accumulator tuple be computed from the previous one and the new list element?
  - How should the return value of `longest_run/2` be computed from the final accumulator value?
- In-class exercise: implement `longest_run/2`.

Preview

January 8: Processes and Messages

Reading: *Learn You Some Erlang*, Higher Order Functions and The Hitchhiker's Guide... and More on Multprocessing

Homework: **Homework 1 goes out (due Jan. 17)** – Erlang programming

Mini-Assignment: Mini-Assignment 1 due **1:00pm**

January 10: Reduce – The Algorithm

Reading: *Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures

---

January 12: Reduce – The Pattern

Reading: Lin & Snyder, chapter 5, pp. 112–125

January 15: Scan

Homework: Homework 1 deadline for early-bird bonus (11:59pm)

Homework 2 goes out (due Oct. 4) – Reduce and Scan

January 17: Reduce & Scan Examples

Homework: Homework 1 due **11:59pm**

September 22: Message Passing Machines

September 25: Cache Coherency

September 27: Programming with Shared Memory

September 29: Data Parallel Machines

October: Performance Analysis and Parallel Algorithms

Review Questions (1 of 6)

- What is referential transparency?
- Why don't functional languages have for-loops or while-loops?
- What is pattern matching?
  - Consider the function `non_decreasing_too_many_ifs(List)` below that returns true iff each element of `List` is greater than or equal to the element that came before it.
  - Write a `non_decreasing` function using pattern matching. `non_decreasing` should produce the same results as `non_decreasing_too_many_ifs`, but the code should be simpler and easier to read.

```
non_decreasing_too_many_ifs(X) ->
  if is_list(X) ->
       if X==[] -> true; % an empty list is non-decreasing
          tl(X) == [] -> true; % a singleton list is non-decreasing
          true ->      (hd(X) =< hd(tl(X)))
                   and non_decreasing_too_many_ifs(tl(X))
       end;
     not is_list(X) ->
       error("non_decreasing_too_many_ifs(X): X is not a list")
  end.
```

Review Questions (2 of 6)

- Some of these are from material in this lecture, some are from material in the assigned readings from *Learn You Some Erlang* (i.e. Introduction through Higher-Order Functions), some are from both.
- What is an Erlang atom? Give an example of a use of an atom.
- If `X` and `Y` are Erlang numbers, what is the difference between `X / Y` and `X div Y` ?
- What is the special role of `_` of `_Name` as the name of an Erlang variable?
- What is the difference between an Erlang list and an Erlang tuple?
  - Write a the expression for the **list** that has the elements `1`, `2`, and `3` in ascending order.
  - Write a the expression for the **tuple** that has the elements `1`, `2`, and `3` in ascending order.
  - Describe the typical uses of lists and tuples.

Review Questions (3 of 6)

- What is a `fun` expression? Give an example.
- What does the Erlang shell print when you type

  [104,101,108,108,111,32,119,111,114,108,100].

  at the Erlang prompt? Why? This is kind-of a trick question. Think about it. Then try it for real in an actual Erlang shell to check your answer.
- Let `neighbours(List)` be a function that returns a list of tuples where each tuple consists of two, successive elements of `List`. For example:

```
neighbours([]) -> [];
neighbours([1]) -> [];
neighbours([1, 2]) -> [{1,2}];
neighbours([1, 2, 3]) -> [{1,2}, {2,3}];
neighbours([1, 2, 3, a, b, c]) ->
  [{1,2}, {2,3}, {3,a}, {a,b}, {b,c}];
```

  Write an implementation of `neighbours`. Of course, you should use pattern matching.
- See also `zip`.

Review Questions (4 of 6)

- Consider the code below for flattening a nested list.
- Write a `flatten` function using pattern matching. `flatten` should produce the same results as `flatten_too_many_ifs`, but the code should be simpler and easier to read.

```
flatten_too_many_ifs(X) ->
  if is_list(X) ->
       if X==[] -> X;
          tl(X) == [] -> X;
          true->
            FlatHead =
            if is_list(hd(X)) -> flatten_too_many_ifs(hd(X));
               true -> hd(X)
            end,
            FlatTail = flatten_too_many_ifs(tl(X)),
            FlatHead ++ FlatTail
       end;
     not is_list(X) ->
       error("flatten_too_many_ifs(X): X is not a list")
  end.
```

Review Questions (5 of 6)

- This lecture described the higher-order function `fold` and mentioned its implementations in the Erlang standard library as `foldl` and `foldr`. Here, we will consider some other higher-order that will make your programming life easier.
- `filter(Pred, List)` returns the elements of `List` for which the function `Pred` returns `true`.
  - Example – select multiples of 3.

```
1> L = seq(1,10).
[1,2,3,4,5,6,7,8,9,10].
2> filter(fun(X) -> (X rem 3) == 0 end, L).
[3,6,9].
```

  - Use `filter` and `length` to implement `count3s`.
- `all(Pred, List)` returns true iff every element of `List` is `true`. If `List` has an element that is not equal to `true`, the first such element must be `false`.
  - Use `all` and `neighbours` to implement `non_decreasing`.
  - See also: `any`.

Review Questions (6 of 6)

- We're going to make a simple calculator. Let

```
calc_help(Acc, {'+', Val})  -> Acc + Val;
calc_help(Acc, {'-', Val})  -> Acc - Val;
calc_help(Acc, {'~', Val})  -> Val - Acc;
calc_help(Acc, {'*', Val})  -> Acc * Val;
calc_help(Acc, {'/', Val})  -> Acc / Val;
calc_help(Acc, {'\\', Val}) -> Val / Acc.

calc_l(Acc0, Ops) ->
  lists:foldl(fun(Op, Acc) -> calc_help(Acc, Op) end,
              Acc0, Ops).

calc_r(Acc0, Ops) ->
  lists:foldr(fun(Op, Acc) -> calc_help(Acc, Op) end,
              Acc0, Ops).
```

- Try

```
erl_intro:calc_l(5, [{'+', 7}, {'/', 3}, {'-', 2}, {'*', 10}]).
erl_intro:calc_r(5, [{'+', 7}, {'/', 3}, {'-', 2}, {'*', 10}]).
```

  - Why do `calc_l` and `calc_r` give different results?
  - Which makes the more "reasonable" calculator?

- Add another operation to the calculator, e.g. remainder, square-root, exponentiation, or anything else that you like.

Supplementary Material

- Erlang resources
- Common issues with lists
- Staying sane in spite of Erlang punctuation
- Erlang atoms
- A few hints for using the Erlang shell

Erlang Resources

- *LYSE* – you should be reading this already!
- Install Erlang on your computer
  - Erlang solutions provides packages for and the most common linux distros
    https://www.erlang-solutions.com/resources/download.html
  - Note: some linux distros come with Erlang pre-installed, but it might be an old version. You should probably install from the link above.
- http://www.erlang.org
  - Searchable documentation
    http://erlang.org/doc/search/
  - Language reference
    http://erlang.org/doc/reference_manual/users_guide.html
  - Documentation for the standard Erlang library
    http://erlang.org/doc/man_index.html
- The CPSC 418 Erlang Library
  - Documentation
    http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/doc/index.html
  - `.tgz` (source, and pre-compiled `.beam`)
    http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz

Remarks about Constructing Lists

It's easy to confuse `[A, B]` and `[A | B]`.

- This often shows up as code ends up with crazy, nested lists; or code that crashes; or code that crashes due to crazy, nested lists; ....
- Example: let's say I want to write a function `divisible_drop(N, L)` that removes all elements from list `L` that are divisible by `N`:

```
divisible_drop(_N, []) -> []; % the usual base case
divisible_drop(N, [A | Tail]) ->
  if A rem N == 0 -> divisible_filter(N, Tail);
     A rem N /= 0 -> [A | divisible_filter(N, Tail)]
  end.
```

It works – see the code in erl_intro.erl.

```
3> erl_intro:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
[1,4,17,100]
```

Misconstructing Lists

Working with `divisible_drop` from the previous slide...

- Now, change the second alternative in the `if` to
  `A rem N /= 0 -> [A, divisible_filter(N, Tail)]`
  Trying the previous test case:

```
4> erl_intro:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
[1,[4,[17,[100,[]]]]]
```

  Moral: If you see a list that is nesting way too much, check to see if you wrote a comma where you should have used a `|`.
- Restore the code and then change the second alternative for `divisible_drop` to `divisible_drop(N, [A, Tail]) ->` Trying our previous test:

```
5> erl_intro:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
** exception error: no function clause matching...
```

Punctuation

- Erlang has lots of punctuation: commas, semicolons, periods, and `end`.
- It's easy to get syntax errors or non-working code by using the wrong punctuation somewhere.
- Rules of Erlang punctuation:
  - Erlang declarations end with a period: `.`
  - A declaration can consist of several alternatives.
    - ★ Alternatives are separated by a semicolon: `;`
    - ★ Note that many Erlang constructions such as `case`, `fun`, `if`, and `receive` can have multiple alternatives as well.
  - A declaration or alternative can be a block expression
    - ★ Expressions in a block are separated by a comma: `,`
    - ★ The value of a block expression is the last expression of the block.
  - Expressions that begin with a keyword end with `end`
    - ★ `case` *Alternatives* `end`
    - ★ `fun` *Alternatives* `end`
    - ★ `if` *Alternatives* `end`
    - ★ `receive` *Alternatives* `end`

Remarks about Atoms

- An atom is a special constant.
  - Atoms can be compared for equality.
  - Actually, any two Erlang can be compared for equality, and any two terms are ordered.
  - Each atom is unique.
- Syntax of atoms
  - Anything that looks like an identifier and starts with a lower-case letter, e.g. `x`.
  - Anything that is enclosed between a pair of single quotes, e.g. `'47 BIG apples'`.
  - Some languages (e.g. Matlab or Python) use single quotes to enclose string constants, some (e.g. C or Java) use single quotes to enclose character constants.
    - ★ But not Erlang.
    - ★ The atom `'47 big apples'` is not a string or a list, or a character constant.
    - ★ It's just its own, unique value.
  - **Atom constants can be written with single quotes, but they are not strings.**

Avoiding Verbose Output

- Sometimes, when using Erlang interactively, we want to declare a variable where Erlang would spew enormous amounts of "uninteresting" output were it to print the variable's value.
  - We can use a comma (i.e. a block expression) to suppress such verbose output.
  - Example

```
6> L1_to_5 = seq(1, 5).
[1, 2, 3, 4, 5].
7> L1_to_5M = seq(1, 5000000), ok.
ok
8> length(L1_to_5M).
5000000
9>
```

Forgetting Bindings (in the shell)

- Referential transparency means that bindings are forever.
  - This can be nuisance when using the Erlang shell.
  - Sometimes we assign a value to a variable for debugging purposes.
  - We'd like to overwrite that value later so we don't have to keep coming up with more names.
- In the Erlang shell, `f(Variable).` makes the shell "forget" the binding for the variable.

```
9> X = 2+3.
5.
10> X = 2*3.
** exception error:  no match of right hand side value 6.
11> f(X).
ok
12> X = 2*3.
6
13>
```

Table of Contents