An Introduction to Erlang

Part 1 – Sequential Erlang

(thanks to Richard Carlsson for the initial version of the slides)

Erlang Buzzwords

- Functional (strict)
- Single-assignment
- Dynamically typed
- Concurrent
- Distributed
- Message passing
- Soft real-time
- Fault tolerant
- No sharing
- Automatic memory management (GC)
- Virtual Machine (BEAM)
- Dynamic code loading
- Hot-swapping code
- Multiprocessor support
- OTP (Open Telecom Platform) libraries
- Open source

Background

- Developed by Ericsson, Sweden
  - Experiments 1982-1986 with existing languages
    - Higher productivity, fewer errors
    - Suitable for writing (large) telecom applications
    - Must handle concurrency and error recovery
  - No good match - decided to make their own
    - 1986-1987: First experiments with own language
    - Erlang (after Danish mathematician A. K. Erlang)
    - 1988-1989: Internal use
    - 1990-1998: Erlang sold as a product by Ericsson
  - Open Source (MPL-based license) since 1998
    - Development still done by Ericsson

Erlang at Uppsala University

- High Performance Erlang (HiPE) research group
  - Native code compiler (SPARC, x86, x86_64, PowerPC, ARM)
  - Program analysis and optimization
  - Runtime system improvements
  - Language development and extensions
  - Programming and static analysis tools
  - Most results from the HiPE project have been included in the official Erlang distribution

Hello, World!

```erl
%% File: hello.erl
-module(hello).
-export([run/0]).
run() -> io:format("Hello, World\n").
```

- `%` starts a comment
- `.` ends each declaration
- Every function must be in a module
  - One module per source file
  - Source file name is module name + `.erl`
- `.` used for calling functions in other modules

Running Erlang

```
$ erl
Erlang (BEAM) emulator version 5.5.1
Eshell V5.5.1 (abort with ^G)
1> 6*7.
42
2> halt().
$$
```

- The Erlang VM emulator is called `erl`
- The interactive shell lets you write any Erlang expressions and run them (must end with `.`)
- The `1>`, `2>`, etc. is the shell input prompt
- The `halt()` function call exits the emulator
Compiling a module

- The "c(Module)" built-in shell function compiles a module and loads it into the system
- If you change something and do "c(Module)" again, the new version of the module will replace the old
- There is also a standalone compiler called "erlc"
  - Running "erlc hello.erl" creates "hello.beam"
  - Can be used in a normal Makefile

Running a program

- Compile all your modules
- Call the exported function that you want to run, using "module:function(...)"
- The final value is always printed in the shell
  - "ok" is the return value from io:format(...)"

A recursive function

- Variables start with upper-case characters!
- ';' separates function clauses
- Variables are local to the function clause
- Pattern matching and guards to select clauses
- Run-time error if no clause matches (e.g., N < 0)
- Run-time error if N is not an integer

Tail recursion with accumulator

- The arity is part of the function name: fac/1≠fac/2
- Non-exported functions are local to the module
- Function definitions cannot be nested (as in C)
- Last call optimization: the stack does not grow if the result is the value of another function call

Recursion over lists

- Pattern matching selects components of the data
- "_" is a "don't care"-pattern (not a variable)
- "[Head|Tail]" is the syntax for a single list cell
- "[]" is the empty list (often called "nil")
- "[X,Y,Z|Tail]" is a list with exactly three elements
- "[X,Y,Z|Tail]" has three or more elements

List recursion with accumulator

- The same syntax is used to construct lists
- Strings are simply lists of character codes
  - "Hello" = "$H, $e, $l, $l, $o" = [72,101,...]
  - "" = []
### Numbers
- Arbitrary-size integers (but usually just one word)
- \#-notation for base-N integers
- \$-notation for character codes (ISO-8859-1)
- Normal floating-point numbers (standard syntax)
  - cannot start with just a '.', as in e.g. C

```erlang
12345
-9876
16#fff
28010101
$A
0.0
3.1415926
6.023e+23
```

### Atoms
- Must start with lower-case character or be quoted
- Single-quotes are used to create arbitrary atoms
- Similar to hashed strings
  - Use only one word of data (just like a small integer)
  - Constant-time equality test (e.g., in pattern matching)
  - At run-time: `atom_to_list(Atom), list_to_atom(List)`

```erlang
true  % boolean
can't start with just a '.
false % boolean
ck   % used as "void" value
hello_world
```

### Tuples
- Tuples are the main data constructor in Erlang
- A tuple whose 1st element is an atom is called a **tagged tuple** - this is used like constructors in ML
  - Just a convention – but almost all code uses this
- The elements of a tuple can be any values
- At run-time: `tuple_to_list(Tup), list_to_tuple(List)`

```erlang
()  % empty tuple
(42) % single element tuple
(1,2,3,4) % multiple element tuple
(movie, "Yojimbo", 1961, "Kurosawa")
(foo, [bar, X]),
    (baz, Y),
    [1,2,3,4,5])
```

### Other data types
- Functions
  - Anonymous and other
- Bit streams
  - Sequences of bits
    - \&\&\{0,1,2,...,255\}\}
- Process identifiers
  - Usually called 'Pids'
- References
  - Unique "cookies"
  - `R = make_ref()`
- No separate booleans
  - atoms `true/false`
- Erlang values in general are often called “terms”
- All terms are ordered and can be compared with <, >, ==, =:=, etc.

### Type tests and conversions
- Note that `is_list` only looks at the first cell of the list, not the rest
- A list cell whose tail is not another list cell or an empty list is called an “improper list”.
  - Avoid creating them!
- Some conversion functions are just for debugging: avoid!
  - `pid_to_list(Pid)`

```erlang
is_integer(X) \%\%
is_float(X) \%
is_number(X) \%
is_atom(X) \%
is_tuple(X) \%
is_pid(X) \%
is_reference(X) \%
is_list(X) \%
```

### Built-in functions (BIFs)
- Implemented in C
- All the type tests and conversions are BIFs
- Most BIFs (not all) are in the module “erlang”
- Many common BIFs are auto-imported (recognized without writing "erlang:...")
- Operators (+,\*,\/,...) are also really BIFs

```erlang
length(List) \%
tuple_size(Tuple) \%
element(N, Tuple) \%
setelement(N, Tuple, Val) \%
abs(N) \%
round(N) \%
trunc(N) \%
throw(Term) \%
halt() \%
time() \%
date() \%
own() \%
self() \%
spawn(Function) \%
exit(Term) \%
```
Standard Libraries

- Application Libraries
  - kernel
  - erlang
  - file, filelib
  - ets
- stdlib
  - lists
  - dict, ordict
  - sets, gb_sets
  - gb_trees
  - gb_sets
  - file, filelib
  - inet
  - os
- stdlib

Written in Erlang

“Applications” are groups of modules

- Libraries
- Application programs
- Servers/daemons
- Tools
- GUI system (gs)

Expressions

- The usual operators
  \[(X + Y) / -Z * 10 - 1\]

- Boolean
  \[X \text{ and not } Y \text{ or } (Z \text{ xor } W)\]

- Bitwise operators
  \[(X \text{ or } Y) \text{ and } Z\]

- Comparisons
  \[X \text{ /= } Y\]

- List operators
  \[List1 ++ List2\]

Fun-expressions

- Anonymous functions (lambda expressions)
  - Usually called “funs”
  - Can have several clauses
  - All variables in the patterns are new
  - All variable bindings in the fun are local
  - Variables bound in the environment can be used in the fun-body

```erlang
F1 = fun () -> 42 end
F2 = fun (X) -> X + 1 end
F3 = fun (X, Y) ->
  X + Y;
  X + Y;
  (X, Y) ->
end
F4 = fun ([foo, X], Y) ->
  X + Y;
  [bar, X], Y) ->
  X + Y;
  (_, Y) ->
end
F5 = fun list/3
F6 = fun mod:4/3
```

Pattern matching with ‘=’

- Tuple = {foo, 42, “hello”},
  (X, Y, Z) = Tuple,

- List = [5, 5, 5, 3, 2, 1],
  [A, B | Tail] = List,

- Struct = {foo, [5, 6, 7, 8}, [17, 42]},
  {foo, [A|Tail], [N, Y]} = Struct

- Match failure causes runtime error (badmatch)
- Successful matching binds the variables
  - But only if they are not already bound to a value!
  - Previously bound variables can be used in patterns
  - A new variable can also be repeated in a pattern

```erlang
case List of
  [X|Xs] when X >= 0 ->
    X + f(Xs);
  [_X|Xs] ->
    f(Xs);
  [] ->
    0;
  _ ->
    throw(error);
end```

Case-switches

- Any number of clauses
- Patterns and guards, just as in functions
- “;” separates clauses
- Use “,” as catch-all
- Variables may also begin with underscore
- Signals “I don’t intend to use this value”
- Compiler won’t warn if variable is not used

```erlang
case List of
  [X|Xs] when X >= 0 ->
    X + f(Xs);
  _X|Xs] ->
    f(Xs);
  [] ->
    0;
  _ ->
    throw(error);
end```

If-switches and guard details

- Like a case-switch without the patterns and the “when” keyword
- Use “true” as catch-all
- Guards are special
  - Comma-separated list
  - Only specific built-in functions (and all operators)
  - No side effects

```erlang
if
  X >= 0, X < 256 ->
  true ->
  f(Xs)"
end```
List comprehensions

- Left of the "||" is an expression template
- "Pattern <- List" is a generator
  - Elements are picked from the list in order
- The other expressions are boolean filters
  - If there are multiple generators, you get all combinations of values

```
%% map
[X || X <- List]
%% filter
[X || X <- Xs, X > 0]
```

Sort example
```
qsорт([P|Xs]) ->
  qsорт([X || X <- Xs, X < P]) ++ [P] ++
  qsорт([X || X <- Xs, X >= P]);
qsорт([]) -> [].
```

Catching exceptions

- Three classes of exceptions
  - `throw`: user-defined
  - `error`: runtime errors
  - `exit`: end process
- Only catch `throw` exceptions, normally (implicit if left out)
  - Re-thrown if no catch-clause matches
  - "after" part is always run (side effects only)

```
try lookup(X)
catch not_found ->
  use_default(X);
exit:Term ->
  handle_exit(Term)
end
%% with 'of' and 'after'
try lookup(X, File) of
  Y when Y > 0 -> f(Y);
  Y -> g(Y)
catch...
after close_file(File)
end
```

Old-style exception handling

```
Val = (catch lookup(X)),
  case Val of
    not_found ->
      %% probably thrown
      use_default(X);
      ('EXIT', Term) ->
        handle_exit(Term);
      _ -> Val
      end
```

Record syntax

```
-record(foo, {a=0, b}.
{foo, 0, 1} = #foo{b=1}
R = #foo{}
{foo, 0, undefined} = R
{foo, 2, 1} = #foo{b=1, a=2}
0 = R#foo.a
undefined = R#foo.b
f(#foo{b=undefined}) -> 1;
f(#foo[a=A, b=B]) when B > 0 -> A + B;
f(#foo[]) -> 0.
```

Preprocessor

```
#include("defs.hrl").
-ifdef(FI).
-define(FI, 3.1415926).
-endif.
area(R) -> ?PI * (R*R).
-define(foo(X), {foo,X+1}).
{foo,2} = ?foo(1)
%% pre-defined macros
-MODULE
-LINE
```

Type declarations

- Erlang has a notation for declaring types out of the "built-in" ones
  - `type fruit() :: 'apple' | 'banana' | 'orange'`
  - `type fruit_list() :: [fruit()].`
  - `type atom_int_list() :: [atom() | integer()].`
- These types can then be used to declare the type of record fields
  - `record(my_rec, {a = 0 :: integer(), b :: fruit()}, c = [] :: atom_int_list()).`
Spec declarations

- Types can also be used to declare the type of function arguments and return type
  ```erlang
  -spec price(fruit()) -> integer().
  price(apple) -> 10;
  price(banana) -> 9;
  price(orange) -> 8.
  ```

- ... and they can be used to impose constraints that are not necessarily present in the code but reflect programmers’ intentions
  ```erlang
  -spec my_app([atom()], [integer()]) -> atom_int_list().
  my_app([], Is) -> Is;
  my_app([A|As], Is) -> [A | my_app(As, Is)].
  ```

Dialyzer

- A static analysis tool that finds discrepancies in Erlang code bases

End

Resources:
www.erlang.org
- Getting Started
- Erlang Reference Manual
- Library Documentation