Erlang

The Good the Bad and the Ugly

Joe Armstrong

- Confusing things
- Things that we got wrong
- Removing things
- Wrong Assumptions
- Stuff that's not bad
- Things without names
- Missing Functionality
- Syntactic Improvements
- Missing BIFs
- Way forward

Confusing things

- Shell input and module text are not the same
- records cannot be (easily) printed

Things that we got wrong

- No updated formal specification
  The only spec is very old and not maintained
- No formal specification of BEAM
  “read the code and comments”
- Cannot remove things

Removing things

- Inheritance

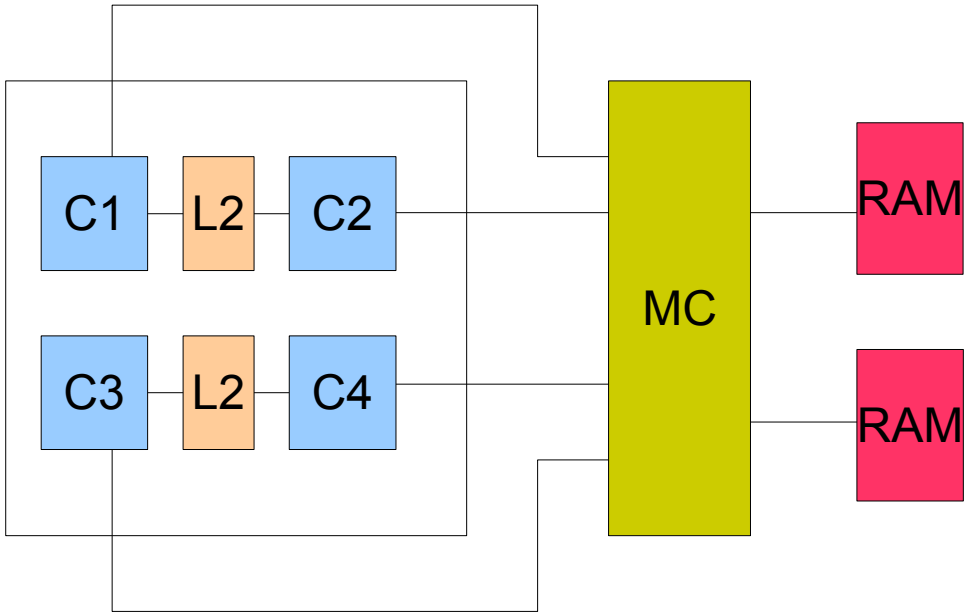
```
-module(abc).
 -extend(old).
 -hide([foo/1,bar/2]).
```

- Language version

```
-module(abc).
 -requires(erl6).
```

Wrong Assumptions

- Same time to send a message between any two processes within the same node. *Not true on multicores.*



Harpertown

Message passing between C1 and C2, or C3 and C4 goes through an L2 cache and is fast. Between C1 and C3 goes through main memory C3 and is slow ...

Stuff that's not bad

- Strings – there is no “string problem”
- Syntax – there is no syntax problem – *except* there is too much syntax

Stuff we could do without

- Macros - ifdefs
- Include files
- Records (use hashmaps)

- Fewer syntactic forms = easier to write
  refactoring tools integrate with eclipse etc.

Stuff we need to think about

- Safety – running millions of processes in an untrusted environment.
- Namespaces *and versioning* for multiple programs running in the same VM
- Fitting things togther – GIT type notions in the language

Missing functionality (in the libaries)

cloud_spawn(Fun)
cloud_store(Key, Val)
fault_tolerent_do(Fun/0)
safe_eval(F/0)
store_forever(key, Val)

Syntactic Improvments

- !! (infix RPC)
- Objective C like positional syntax
- String prefixes in string literals regexp”....”
  Z = xml”<hello>joe</hello>”
- Hashmaps
  X = <{a:123, b:hello, c:dog},
  Y = <{b:joe | X }>,
  <{ a:P | Q }> = Y ...

  P = 123, Q=<{b:joe, c:dog}>

Missing BIFs

Bifs do things you cannot do in code

- atom_to_list(Atom)
- list_to_tuple(L)

Often converting internal representations (types) which are otherwise inaccessable.

For N types we need N*N bifs or 2*N if we implement type_to_list(T) and list_to_type(L)

BIFs

```
1 > if is_atom(abc) -> yes end.
yes
2> atom_to_list(abc).
“abc”

3> if is_tuple({a,b,c}) -> yes end.
yes
4> tuple_to_list({a,b,c}).
[a,b,c]

5> if is_float(3.14159) -> yes end.
yes
6> float_to_list(3.14159).
"3.14158999999999988262e+00"

7> if is_fun(fun(X) -> 2*X end) -> yes end.
????
```

Bifs

```
1> if is_fun(fun(X) -> 2*X end) -> yes
 end.
*1 : illegal guard expression
2> if is_function(fun(X) -> 2*X end) ->
 yes end.
* 1: illegal guard expression
3> D = fun(X) -> 2*X end.
#Fun<erl_eval.6.13229925>
4> if is_function(D) -> yes end.
yes
```

Bifs

```
3> D = fun(X) -> 2*X end.
#Fun<erl_eval.6.13229925>
4> if is_function(D) -> yes end.
Yes
5> L = erlang:
 erlang:fun_to_list(D).
"#Fun<erl_eval.6.13229925>
6> erlang:list_to_fun(...) %%
 missing
```

What I'd like

```
1> D = fun(X) -> 2*X end.
#Fun<erl_eval.6.13229925>
2> if is_fun(D) -> yes end.
yes
3> L = fun_to_list(D).
"fun(X) -> 2 * X end"
4> list_to_fun(L).
#Fun<erl_eval.6.13229925>
```

Symmetry

For every X there should be an X^{-1}

Introspection

```
-module(foo).
-export([plus/2]).

plus(X,Y) -> X + Y.

> fun_to_list(fun foo:plus/2).
“plus(X,Y) -> X+Y.”
```

Module Types

```
> L = [{double, fun(X) -> 2*X   end},
       {plus, fun(X,Y) -> X + Y end},
       ...].
> M = list_to_module(L).
#Mod<31478294>
> M:double(3).
6
> register(glurk, M).
glurk:double(4).
8
> module_to_list(M).
[{double, #Fun<28171836>}, ...]
```

The magic of register

```
Pid = spawn(fun() -> ... end)
register(abc, Pid)
abc ! Msg
```

safe

convenient

```
Fun = fun(X, Y) -> X + Y end,
register(plus, Fun)
plus(2, 3)

Mod = list_to_module(
       [{plus,
         fun(X,Y) -> X + Y end},...]
Mod:plus(2,3)
register(mymod, Mod)
mymod:plus(2,4)
```

Things without names

If things don't have names it is difficult to talk about them.

- Processes - “the processes created by evaluating spawn(fun() -> loop() end”
- Protocols – we use them a lot but never name nor define them

Naming processes

```
-module(foo).
+process(counter/1).
counter(N) -> ...

Pid = newProcess counter(N).

Means

Pid = spawn(fun() -> counter(N) end).
```

Naming Protocols

```
-protocol(fileServer).
start(A) ->
    receive
        A ? listFiles ->
            A ! [file::string()],
            start(A);
        A ? {get, file::string()} ->
            A ! {yes, file::bin()} |
                eNoFile},
            start(A)
     end.

Pid = newProtocol(fileServer, ...)
```

Program development

- All programs are derived from the “ur-program”

```
transform(P1, T) -> P2
cloan(P1) -> P2
merge(P1, P2, D) -> P3
```

- Keep a record of all transformations
  replay the sequence (Smart GIT)

- Interact through web browser
  HTML5/contentEditable/AJAX/xslt-fo

The way forward

- Short term
- Medium Term
- Long Term

Short – term
Low hanging fruit

- Shell history preserved over sessions
- xref warnings at compile time
- Fix records (ie fix problems with consistency and printing)
- Improved documentation handling

Medium – term
Some thought required

- Structs, introspection, safety, large scale distribution
- Refactoring in the browser, distributed program development

Long term

- Complete version control – all the bits fit togther properly
- Safety, accountability mechanism inbuilt
- Programs run forever (or until stopped)
- Data stored forever
- Can find stuff