The Erlang Rationale

Robert Virding

A Rationale

Rationale – n. 1. Fundamental reasons; the basis. 2. An exposition of principles or reasons.

- Why would we want one?

- Help users understand how/why to use various features
- Help language designers
- Help implementors
- Help people wishing to extend language

First principles

- High level language to get real benefits.
- Lightweight concurrency
  - The system should be able to handle a large number of processes, process creation, context switching and inter-process communication must be cheap and fast.
- Asynchronous communication
- Process isolation
  - We don’t want what is happening in one process to affect any other process.
- Error handling
  - The system must be able to detect and handle errors.
- Continuous evolution of the system
  - We want to upgrade the system while it is running and with no loss of service.

First principles

- The language should be simple
  - Simple in the sense that there should be a small number of basic principles, if these are right then the language will be powerful but easy to comprehend and use. Small is good.
  - The language should be simple to understand and program.
- We should provide tools for building systems not solutions
  - We would provide the basic operations needed for building communication protocols and error handling.

Trivial code example

```
ringing_a_side(Addr, B_Pid, B_Addr) ->
    receive
        on_hook ->
            B_Pid ! cleared,
            tele_os:stop_tone(Addr),
            idle(Addr);
        answered ->
            tele_os:stop_tone(Addr),
            tele_os:connect(Addr, B_Addr),
            speech(Addr, B_Pid, B_Addr);
        {seize,Pid} ->
            Pid ! rejected,
            ringing_a_side(Addr, B_Pid, B_Addr);
        _ ->
            ringing_a_side(Addr, B_Pid, B_Addr)
    end.
```

Trivial code example

```
ringing_b_side(Addr, A_Pid) ->
    receive
        cleared ->
            tele_os:stop_ring(Addr),
            idle(Addr);
        off_hook ->
            tele_os:stop_ring(Addr),
            A_Pid ! answered,
            speech(Addr, A_Pid, not_used);
        {seize,Pid} ->
            Pid ! rejected,
            ringing_b_side(Addr, A_Pid);
        _ ->
            ringing_b_side(Addr, A_Pid)
    end.
```

Things missing in early Erlang

- Code handling
- Funs
- ETS
- Binaries
- OTP

Erlang ”things”

- Only two basic types of things in Erlang

- Immutable data structures
  - Normal Erlang terms
- Processes
  - Everything with internal state

- Yes, the process dictionary is a mutable data structure, but we never really liked it!

Processes

- A process is something which obeys *process semantics*:
  - Communicates through asynchronous message passing
  - Links/monitors for error detection/handling
  - Obey/transmit exit signals
  - Parallel independent execution
- N.B. Implementation and internal details irrelevant!

Process communication

- ***All*** process communication by messages
- ***All*** process communication asynchronous

- Process BIFs asynchronous
  - Only check arguments
  - One exception then: sending to registered name!
- Works the same with distribution!

Ports

- ”Processes” for communicating with the outside world
- Obey process semantics
  - Message based interface
  - Obeys links and exit signals
  - Fits in with rest of erlang
- Ports processes on the outside which talk to hardware
- We viewed hardware as being ”active”
- Ports need connected process to communicate with.

Errors and error handling

- Added as ”easy” way to build robust systems
- Allow critical robust core to handle unsafe user code
- Follows process oriented system design
- Co-exists with rest of concurrency, very asynchronous
- Simple bi-directional state version fine for original systems but not sufficient
- Provide the tools not the solution

Modules and code

- Erlang system always been compiled since leaving Prolog
- Erlang modules very basic, only have a name and exported functions
- All functions belong to module
- Module basis for code handling and compilation, easier that way
- Multiple versions needed to do controlled upgrading
- Why 2 versions? Why not? And more explicit versions becomes difficult to handle

I/O-system and servers

- i/o-server between app and i/o-device/port
- Must be a process so all processes in app can use it
- Handles mapping i/o-requests from apps to ports
- Allows generic i/o-functions as i/o-server handles device specifics
- Means i/o-server is generic as i/o-functions handle specific requests

Process groups

- Erlang like an OS

→Should be possible to run many apps at same time

- Processes in an app would need specific ”system” information

→Created process groups

Each group has a group leader

A group is all processes with the same leader

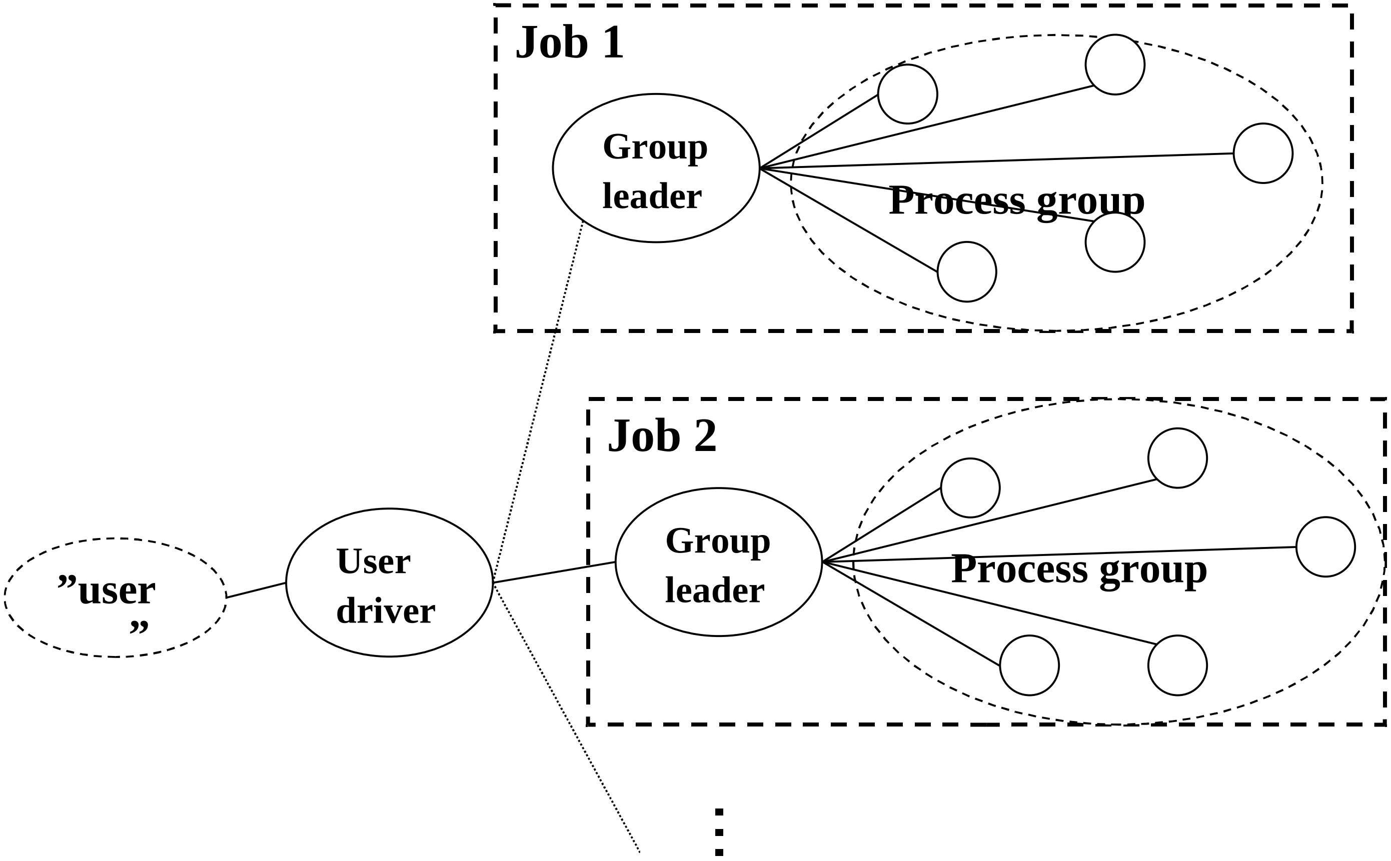
Jobs and the JCL

- One problem with running many apps is that i/o can become very jumbled

→Solution was to add concept of a ”job” and a user driver.

User driver controls which job communicates with user.

Jobs and the JCL



Pattern-matching and guards

- Pattern matching is a Big Win
- The pattern to match and pull apart data should look the same as pattern to build it
- Guards are ***tests*** providing simple extension to pattern matching
- Guard ***tests*** are not expressions!
- Allowing full boolean expressions is both good and bad

Variables, scoping and =

- Variables are just bind-once references to values
- Also inherited Prologs scoping, or rather lack of scoping, a variable’s scope is the whole function clause
- Affects pattern matching as already occurring variables means testing existing value
- = started its life as simple assignment
- Practical to use it to pull apart return values

Records

- Records added to solve problem of:
  - Named fields in tuples
  - Same efficency as element/setelement
- We decided to use tuples instead of adding new data type.

→Compile-time feature

Lack of explicit typing means record type must always be included

Setting field is not compatible with =

```
X#person.name = ”Robert”
```

can never mean what people would like

Macros

- Originally added to provide named constants
- Arguments and conditional compilation added
- Having them token based allows you to do wonderful and terrible things.
- I still wish that I had done them more lisp-like instead of C-like. (but this is a real pain with complex syntax!)

if

- Originally there was only function matching
- Then **`case`** was added, very practical but a bit naughty.
- However sometimes got cases like this:

```
case 1 of
    _ when ... -> ... ;
    _ when ... -> ...
end
```

→Added **`if`** as quick fix, easy to do as only used guards.

Not used much so we never realized the trouble it would cause.

Characters and strings

- Inherited integers and lists from Prolog
- I like using lists for strings
  - powerful data structure
  - easy to work with
- A char type probably not wrong

Never-ending discussions

- Modules as objects?
- Always generate exceptions for errors?
- Add variable scoping and **`let`**?
- Do somethin about **`if`**. Add **`cond`**?