ErlangRT

A new BEAM virtual machine in Rust

Dmytro Lytovchenko

Sr Developer @ Erlang Solutions, Sweden

@kvakvs

About Me

Was born in Ukraine

A cat person

Senior Developer in Erlang Solutions Sweden

Almost 9 years of Erlang & backend experience

20 years of C and C++ experience

Some game development background, Python, some Ruby, and some old PHP experience

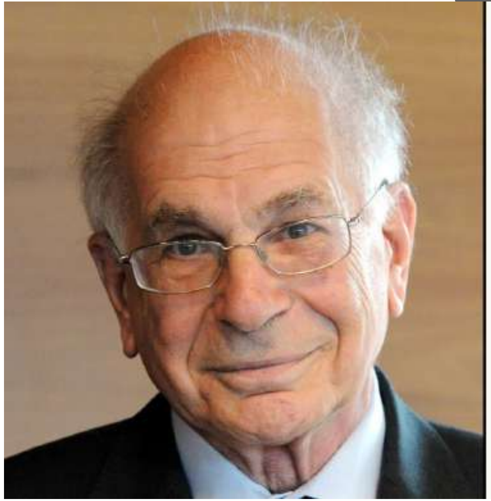
On Being a Human...

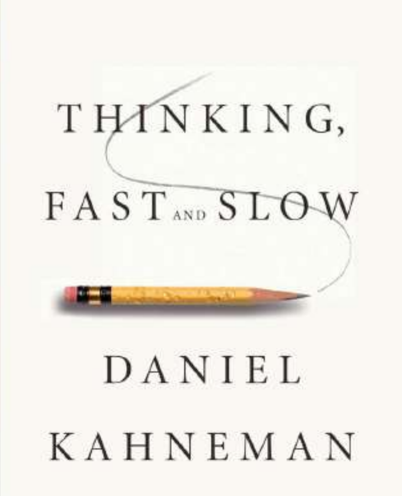
On Being a Human

Daniel Kahneman psychologist and economist (Nobel prize: Economics, 2002)
His book (2011) became a best seller

Human mind runs 2 systems:

- 1 — Automatic, provides solutions
  - Quick ideas without actual evaluation
- 2 — Slow, expensive, clumsy, smart
  - Evaluates, can change or accept the solution





THINKING,
FAST AND SLOW
DANIEL
KAHNEMAN

The Monkey: Fast, automatic, frequent, emotional, stereotypic, unconscious

- Quick comparisons
- Accessing short term memory
- Emotional reactions, disgust, anger etc
  - A typical social app user scrolling
- Familiar activities: walking, cycling, driving with no traffic
- Understanding simple sentences

The Thinker: Slow, effortful, infrequent, logical, calculating, conscious

- Planning your schedule or your actions
- Focusing on something in a noisy env, on a party
- Remembering/recognising a sound
- Finding objects/persons with some property
- Tight parking
- Slow-paced games vs hard opponents
- Calculating numbers or logical reasoning

"Sponge Bob Square Pants" animated series, created by UPP/Nickelodeon, distributed by Viacom

On Procrastination

An article at waitbutwhy.com from 2013
“Why procrastinators procrastinate”

- Instant gratification Monkey
- Panic deadline Monster which scares the Monkey away

Максим
ДОРОФЕЕВ
ДЖЕДАЙСКИЕ ТЕХНИКИ
Как воспитать свою обезьяну, опустошить инбокс и сберечь мыслетопливо

Having More Work Done

An obvious question: How to do more things before getting tired?

The answer: Allow the Monkey to drive more.

- Remove the Monkey food (distractions, social feeds, noise etc)
- Plan your actions in Thinking mode, create checklists with steps to perform, then let the Monkey execute
- Allow yourself some controlled rest
- Allow the Monkey to do mistakes and reveal them to be fixed

Monkey-Friendly Code

- A checklist is planned ahead
- **Let the Machine do the thinking**
- **Let the Machine do the remembering**
- **Let the Machine find the stuff you've missed**
- **Let the Monkey drive the Machine**

} The Solution is well known

Animal-Friendly Coding

C is Not Animal-Friendly

Working with **`erlang/otp`** (C code) requires a lot of focus:

- Everything is either an integer, a pointer or a struct
- Byte sizes, word sizes, bit sizes are all integers
  - Sometimes signed
- Requires locking, often in certain order
- Pre-required knowledge (e.g. construction/call order)
- Naming styles are inconsistent (more remembering)

C is Not Animal-Friendly

- Time is spent making sure that your program is correct
- This distracts us from the actual problem we’re solving

C++ isn’t friendly either

Notable Evolutional Steps of Erlang/OTP

- The new competitor projects are doomed to repeat the history, same evolution steps have to be taken:
  - Single CPU → SMP
  - Choice of GC algorithm
  - Evolution of the BEAM loader and the interpreter loop
- We know the winning strategy
- Possible evolution path: Oxidize the OTP C source?

Why People Choose C?

- Multiple freedoms
- Unsafety is welcome
- “I know what I am doing”
- C runs everywhere
- C is plain simple

Why Not Join Writing the VM in C?

- Limitations (or lack thereof) and unsafety of C
- OTP source is C98 (is there an upgrade plan?)
- OTP source is weakly typed, a lot of conventions you have to remember
- Convoluted build system
- Resistance to major changes (understandable)
    - Ericsson's customers want their code to keep working
- Minor changes take a lot of effort

Why a VM rewrite?

- Discover and publish the missing arcane knowledge about the VM
- Serious challenge, but doable
- Get rid of ancient code, new algorithms, cleanup
- Fun

http://**beam-wisdoms**.clau.se/

github.com/happi/**theBeamBook**

www.erlang-solutions.com

Enter ErlangRT

History of ErlangRT

- 2015 — kvakvs/**GluonVM** in C++ (now abandoned)
- kvakvs/**E4VM** — a prototype for embedded which used Forth E4 VM as inspiration but the goal was to run the BEAM
- Multiple approaches to translate BEAM into simpler bytecode
  - A translator in Haskell producing compressed bitcode
- September 2017 — ErlangRT was started in Rust

Competitor projects

- cloudozer/**ling**
  - Written in C, last commit 4 years ago
  - The selling point was low cost of starting new VMs on the hypervisor
- bettio/**AtomVM**
  - Embedded-oriented VM written in C
  - Finally a practical man doing The Real Thing (unlike my previous approaches)
- archSeer/**enigma**
  - A direct competitor project, in Rust, started in Dec 2018

Rust is Awesome

- Hindley-Milner type system, pattern matching
  - Type inference
- Superior error handling
  - Runtime errors are a pleasure to read
- Minimises the time spent in gdb/gede
- Minimise copy & paste: powerful yet sane macros

Why a VM in Rust?

- Zero-cost abstractions
- Safety is the default
  - Range checking, thread data-sharing rules, etc
  - Unsafety is explicit
- Strong H-M typing:
  - Easy refactoring
  - Everything can have its own type
- Fast as C, portable
  - (Downside) No “goto *label” statement

“How hard could it be?”

1. Decode a BEAM file
2. Implement a few data types
3. A dumb heap with a stack
4. Interpret a few opcodes

...

Is this enough?

- For a prototype maybe, yes

A Fractal of Complexity

Binary Building

- Opcodes for building a binary
  - Insert binary, bitstring, integer, float
    - Process-level context (current binary)
    - Insert from 0th bit, from uneven bit position, into 0th bit, into uneven position
      - Fits into a single byte?
      - Big/little endian? Negative?
      - Does arch support byte addressing?
        - Are source and destination word aligned?

A Fractal of Complexity

Does your VM support binaries?

- Hmm, it might need…
  - Binary term type, binary on bin heap, binary on process heap, refcounted binary, type test op, binary reading ops, binary building ops, binary matching and decomposing ops (big and little endian, signed and unsigned, byte alignment, remember?), search & matching (Aho-Corasick or similar), binary_to_X conversions, X_to_binary conversions, GC support for binaries, binary copy, iolist support… and more

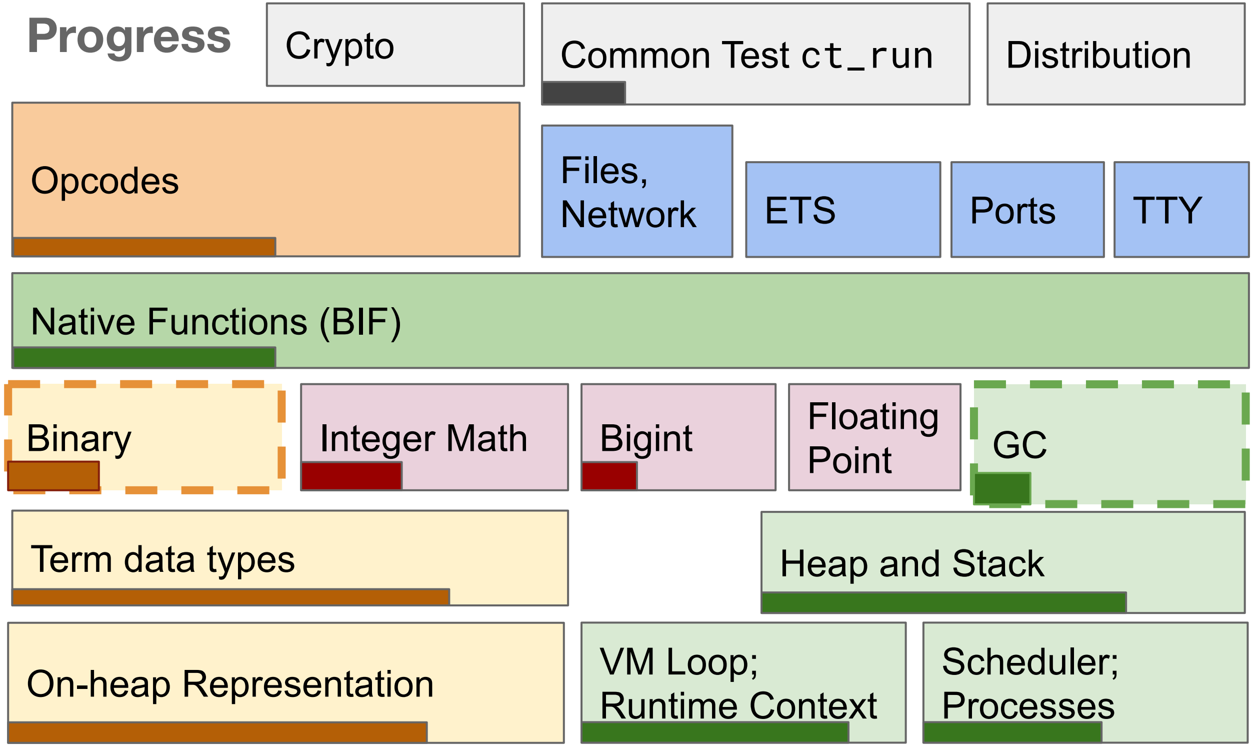
ErlangRT: Goals and Expectations

- Run most of the existing code (no NIF)
- Support at least Linux x64, but minimise usage of **std**
- Support most important VM features
- Have a decent GC algorithm
- Use **erlang/otp** test suites for testing

Progress

- Term data type 80% (remaining 80% are in progress)
- External Term Format — decoder 80%, no encoder
- BEAM Loader — usable
- VM and processes — 40%
- VM loop and opcodes — 45% (74 of 168)
- Some basic BIFs — <15%
- Binaries, sub-binaries, binary heap, binary opcodes — 40%
- GC — started work
- ETS, sockets, ports — not started

Progress



Crypto

Common Test ct_run

Distribution

Opcodes

Files, Network

ETS

Ports

TTY

Native Functions (BIF)

Binary

Integer Math

Bigint

Floating Point

GC

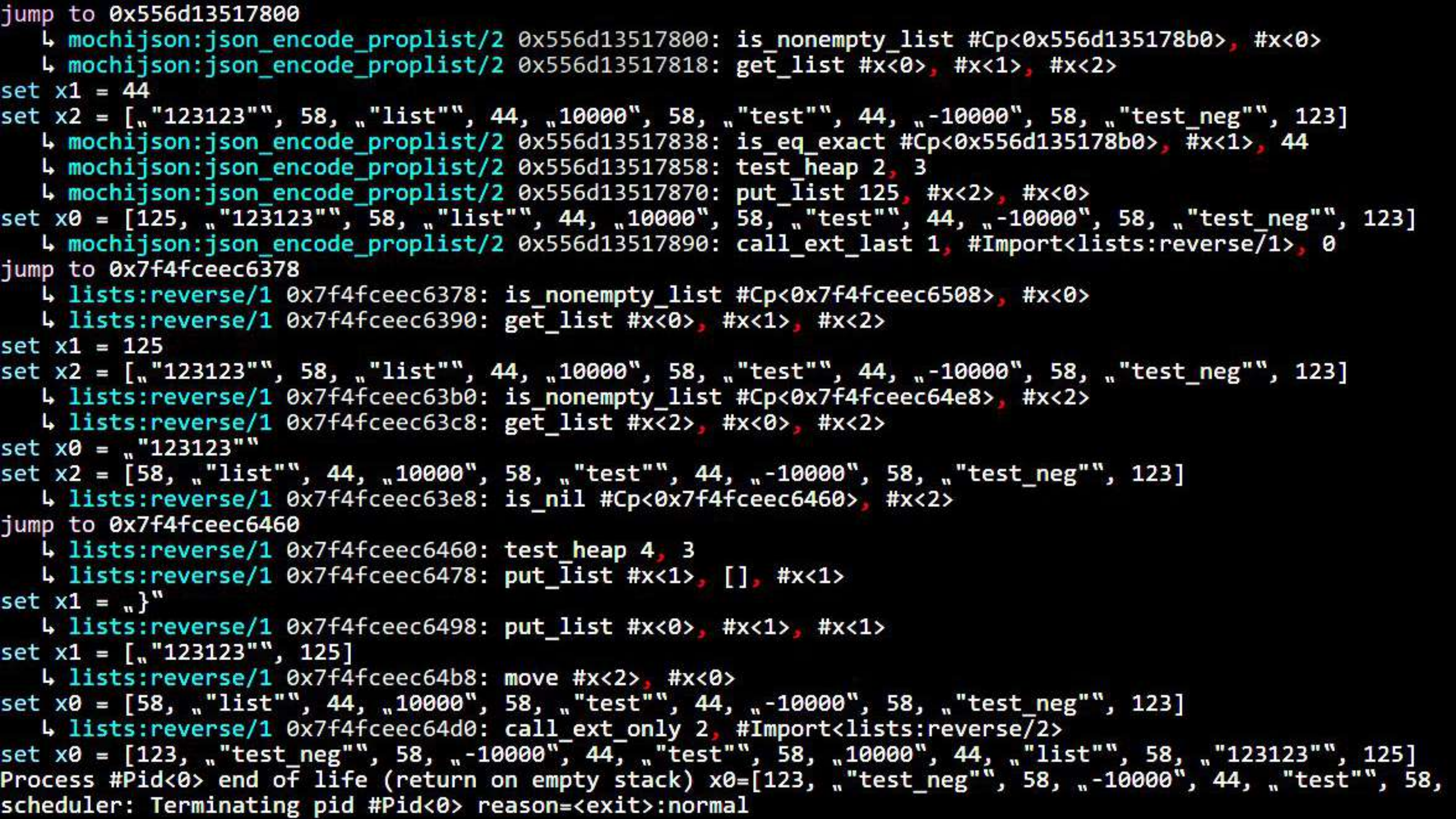
Term data types

Heap and Stack

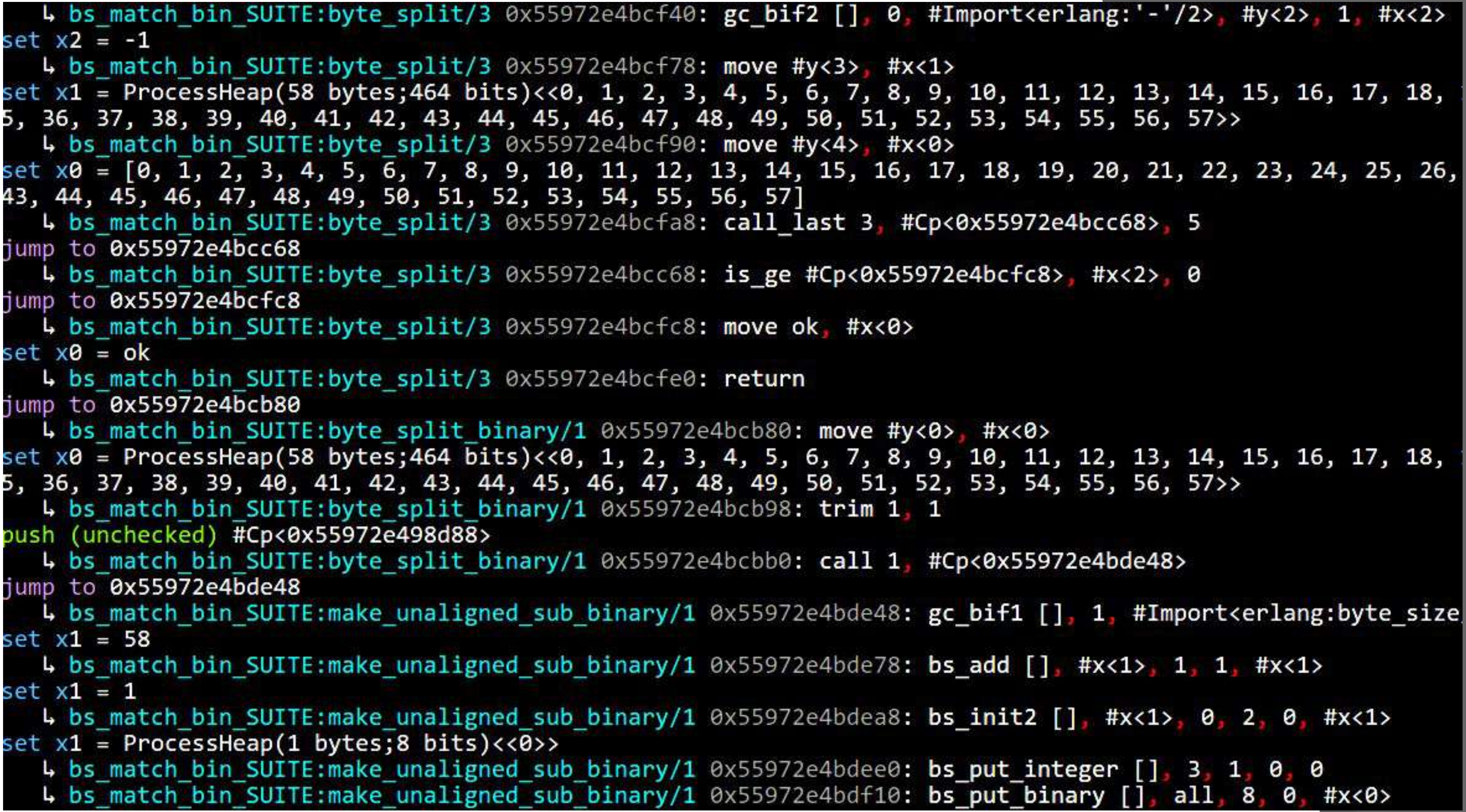
On-heap Representation

VM Loop; Runtime Context

Scheduler; Processes



```
jump to 0x556d13517800
   ↳ mochijson:json_encode_proplist/2 0x556d13517800: is_nonempty_list #Cp<0x556d135178b0>, #x<0>
   ↳ mochijson:json_encode_proplist/2 0x556d13517818: get_list #x<0>, #x<1>, #x<2>
set x1 = 44
set x2 = [„"123123"“, 58, „"list"“, 44, „10000“, 58, „"test"“, 44, „-10000“, 58, „"test_neg"“, 123]
   ↳ mochijson:json_encode_proplist/2 0x556d13517838: is_eq_exact #Cp<0x556d135178b0>, #x<1>, 44
   ↳ mochijson:json_encode_proplist/2 0x556d13517858: test_heap 2, 3
   ↳ mochijson:json_encode_proplist/2 0x556d13517870: put_list 125, #x<2>, #x<0>
set x0 = [125, „"123123"“, 58, „"list"“, 44, „10000“, 58, „"test"“, 44, „-10000“, 58, „"test_neg"“, 123]
   ↳ mochijson:json_encode_proplist/2 0x556d13517890: call_ext_last 1, #Import<lists:reverse/1>, 0
jump to 0x7f4fceec6378
   ↳ lists:reverse/1 0x7f4fceec6378: is_nonempty_list #Cp<0x7f4fceec6508>, #x<0>
   ↳ lists:reverse/1 0x7f4fceec6390: get_list #x<0>, #x<1>, #x<2>
set x1 = 125
set x2 = [„"123123"“, 58, „"list"“, 44, „10000“, 58, „"test"“, 44, „-10000“, 58, „"test_neg"“, 123]
   ↳ lists:reverse/1 0x7f4fceec63b0: is_nonempty_list #Cp<0x7f4fceec64e8>, #x<2>
   ↳ lists:reverse/1 0x7f4fceec63c8: get_list #x<2>, #x<0>, #x<2>
set x0 = „"123123"“
set x2 = [58, „"list"“, 44, „10000“, 58, „"test"“, 44, „-10000“, 58, „"test_neg"“, 123]
   ↳ lists:reverse/1 0x7f4fceec63e8: is_nil #Cp<0x7f4fceec6460>, #x<2>
jump to 0x7f4fceec6460
   ↳ lists:reverse/1 0x7f4fceec6460: test_heap 4, 3
   ↳ lists:reverse/1 0x7f4fceec6478: put_list #x<1>, [], #x<1>
set x1 = „}“
   ↳ lists:reverse/1 0x7f4fceec6498: put_list #x<0>, #x<1>, #x<1>
set x1 = [„"123123"“, 125]
   ↳ lists:reverse/1 0x7f4fceec64b8: move #x<2>, #x<0>
set x0 = [58, „"list"“, 44, „10000“, 58, „"test"“, 44, „-10000“, 58, „"test_neg"“, 123]
   ↳ lists:reverse/1 0x7f4fceec64d0: call_ext_only 2, #Import<lists:reverse/2>
set x0 = [123, „"test_neg"“, 58, „-10000“, 44, „"test"“, 58, „10000“, 44, „"list"“, 58, „"123123"“, 125]
Process #Pid<0> end of life (return on empty stack) x0=[123, „"test_neg"“, 58, „-10000“, 44, „"test"“, 58,
scheduler: Terminating pid #Pid<0> reason=<exit>:normal
```



```
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcf40: gc_bif2 [], 0, #Import<erlang:'-'/2>, #y<2>, 1, #x<2>
set x2 = -1
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcf78: move #y<3>, #x<1>
set x1 = ProcessHeap(58 bytes;464 bits)<<0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 
5, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57>>
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcf90: move #y<4>, #x<0>
set x0 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57]
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcfa8: call_last 3, #Cp<0x55972e4bcc68>, 5
jump to 0x55972e4bcc68
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcc68: is_ge #Cp<0x55972e4bcfc8>, #x<2>, 0
jump to 0x55972e4bcfc8
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcfc8: move ok, #x<0>
set x0 = ok
   ↳ bs_match_bin_SUITE:byte_split/3 0x55972e4bcfe0: return
jump to 0x55972e4bcb80
   ↳ bs_match_bin_SUITE:byte_split_binary/1 0x55972e4bcb80: move #y<0>, #x<0>
set x0 = ProcessHeap(58 bytes;464 bits)<<0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 
5, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57>>
   ↳ bs_match_bin_SUITE:byte_split_binary/1 0x55972e4bcb98: trim 1, 1
push (unchecked) #Cp<0x55972e498d88>
   ↳ bs_match_bin_SUITE:byte_split_binary/1 0x55972e4bcbb0: call 1, #Cp<0x55972e4bde48>
jump to 0x55972e4bde48
   ↳ bs_match_bin_SUITE:make_unaligned_sub_binary/1 0x55972e4bde48: gc_bif1 [], 1, #Import<erlang:byte_size
set x1 = 58
   ↳ bs_match_bin_SUITE:make_unaligned_sub_binary/1 0x55972e4bde78: bs_add [], #x<1>, 1, 1, #x<1>
set x1 = 1
   ↳ bs_match_bin_SUITE:make_unaligned_sub_binary/1 0x55972e4bdea8: bs_init2 [], #x<1>, 0, 2, 0, #x<1>
set x1 = ProcessHeap(1 bytes;8 bits)<<0>>
   ↳ bs_match_bin_SUITE:make_unaligned_sub_binary/1 0x55972e4bdee0: bs_put_integer [], 3, 1, 0, 0
   ↳ bs_match_bin_SUITE:make_unaligned_sub_binary/1 0x55972e4bdf10: bs_put_binary [], all, 8, 0, #x<0>
```

ErlangRT: Plans for 2019-2020

- GC
- Run the **init** and enter the shell
  - Ports/TTY
  - File API
- Common test via **ct_run**

Contributions

- Rust favours large refactoring therefore…
  - The codebase is in constant churn
    - Read some code
    - Join, if you feel brave enough
- Small contributions are the best

Thank you

github.com/kvakvs/**ErlangRT**

More knowledge:

http://**beam-wisdoms**.clau.se/

github.com/happi/**theBeamBook**

@kvakvs