Where are we on the Map$_S$?

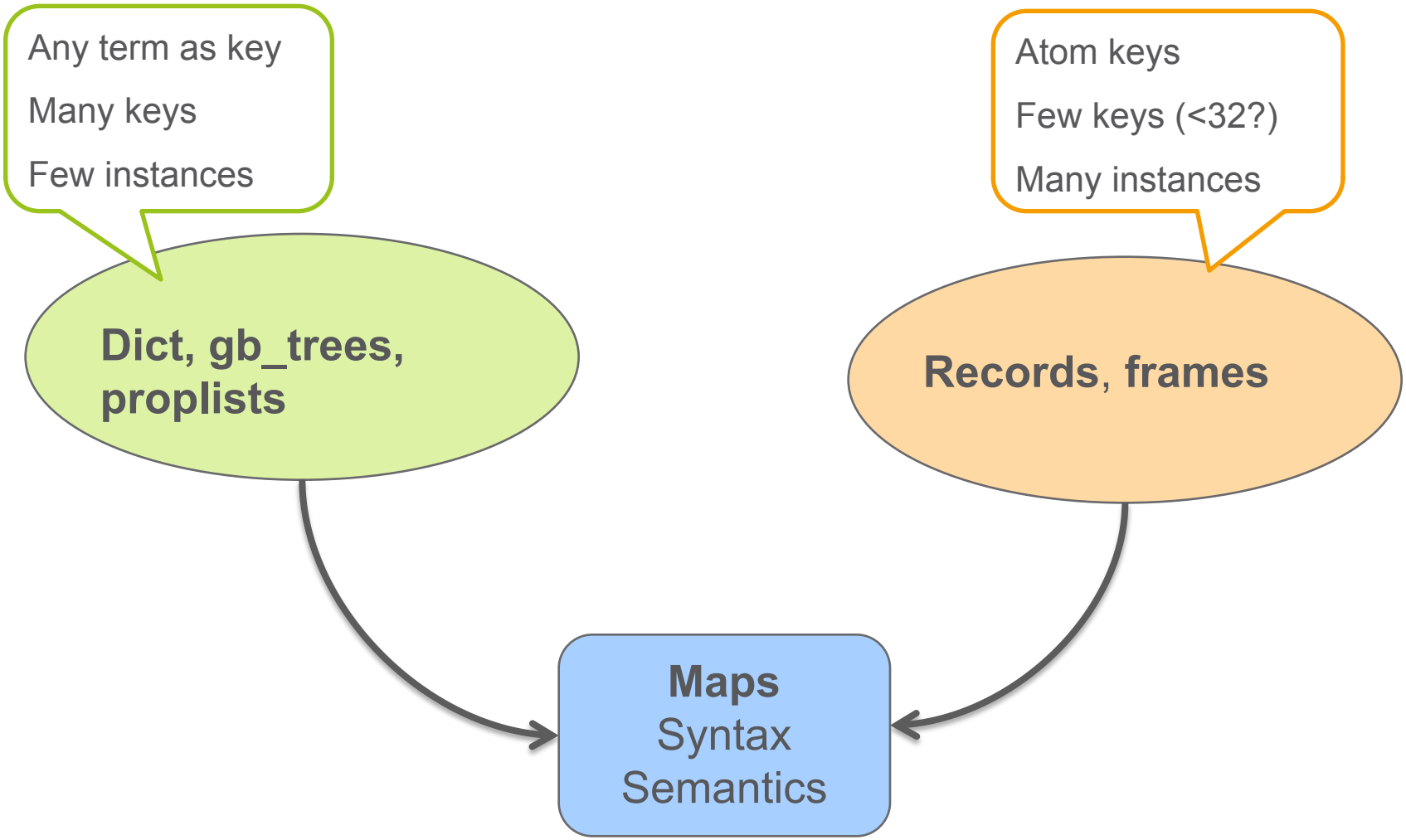
Kenneth Lundin
Erlang/OTP Ericsson

ERICSSON

Why introduce MAPs?

- Maps have been found very useful in other languages such as Perl, Ruby, Python
- Maps in Erlang can give an extra edge when combined with pattern matching
- An easy to use alternative to records, dicts, gb_trees, ets and proplists.
- A complement to records which can superceede them where suitable.

Approach from 2 directions



Goals

- Provide a set of mappings between keys and values which can be easily constructed, accessed and updated
- A data type that can be uniquely distinguished from other data types
- A defined order of key value pairs
- No compile time dependency
- A one to one mapping between parsing and printing the data type.

Syntax: Creation

A map with a single key value pair

```
#{ K => V }
```

<{ K ~ V }> or #{ K = V }

A map with multiple associations:

```
#{ K1 => V1, .. Kn => Vn }
```

An empty map:

```
#{}
```

Examples

```
M1 = #{ a => <<"hello">> }, % single association
M2 = #{ 1 => 2, b => b },   % multiple associations
M3 = #{ A => B },           % single association, variables
M4 = #{ {A, B} => f() }.    % compound key => evaluated
expression
```

Syntax: Update

Similar to create

An expression for **the map to be updated** is put in front of the expression defining the keys to be updated and their respective values.

```
M#{ K => V }
```

Examples:

```
M1 = M0#{ a => 1 },
M2 = M1#{ b => 2, c => 3 },
M3 = M2#{ "function" => fun() -> f() end }.
```

Pattern Matching

Matching of literals as keys are allowed in a function head.

```
handle_call(start, From, #{} = S) ->
    ...
    {reply, ok, S#{ state => start };

%% change only when started
handle_call(change, From, #{ state => start } = S) ->
    ...
    {reply, ok, S#{ state => changed} };
```

Pattern Matching continued

More matching syntax, calculating frequency of terms in a list.
The key is a variable that gets bound in the function head.

```
freq(Is) -> freq(Is, #{}).

freq([I|Is], #{I => C} = M) ->
    freq(Is, M#{ I => C + 1});

freq([I|Is], M) ->
    freq(Is, M#{ I => 1 }); % A new key I is created

freq([], M) ->
    maps:to_list(M).
```

Pattern Matching continued

File information example

Old API's using records or property lists could be refined to use map syntax:

```
1> {ok, #{ type => Type, mtime => Mtime }} =
                                    file:read_file_info(File).
2> io:format("type: ~p, mtime: ~p~n", [Type, Mtime]).
type: regular, mtime: {{2012,7,18},{19,59,18}}
ok
3>
```

Single Value Access

Open Question: Do we need to have single access or is matching sufficient?

```
Value = Map#{Key}.
```

The value is another map!

Examples:

```
1> M = #{ a => 1, c => #{x => 1, y => 7} },
#{x => 1, y => 7} = M#{c}.

2> X = M#{c}#{y}.
7
M1 = M#{c}#{y => 4711}}    % update nested
#{a => 1, c => #{x => 1, y => 4711}}
```

Map Comprehensions

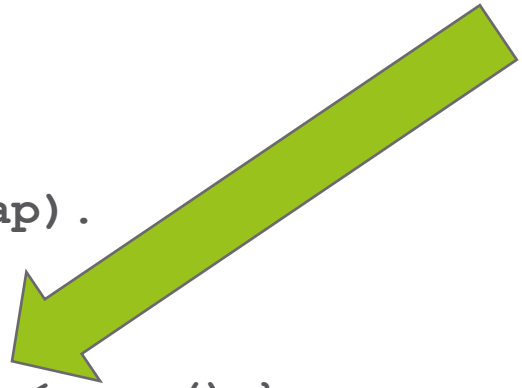
Open Question: The syntax, do we need them?

Generator declaration:

```
K => V <- Map
```

Generator semantics:

```
{K,V} <- maps:to_list(Map).
```



Examples:

```
M0 = #{ K => V*2  || K => V <- map() },
M1 = #{ I => f(I) || I <- list() },
M2 = #{ K => V    || <<L:8,K:L/binary,V/float>> <= binary() }.
```

Type specs

Open Question: Exact syntax, what can Dialyzer take into account?

```
-spec func(M) -> #{ 'opt' = Opt, 'c' = integer() } when
        M   :: #{ 'opt' = Opt, 'c' = integer() },
        Opt :: 'inc' | 'dec'.
func(#{opt => inc, c => C} = M) -> M#{ a => C + 1};
func(#{opt => dec, c => C} = M) -> M#{ a => C - 1}.

-spec plist_to_map(Ls) -> #{ binary() => integer() } when
        Ls :: [{binary(), integer()}].

plist_to_map([{K,V}|Ls], M) when is_binary(K), is_integer(V) ->
   M#{ K => V };
plist_to_map([], M) ->
   M.
```

MAPs are Ordered

- Maps in Erlang are ordered, **Important**!!!!
  - Maps with the same set of keys are always presented in the same way
- The Map data type is sorted after tuple
- Two different maps M1 and M2 are sorted first after size and secondly after their Key=>Value pairs.
- The expression below illustrates:

```
lists:sort([list_to_tuple(maps:to_list(M))
|| M <- [M1,M2]).
```

Functions

Guard BIFs

```
erlang:is_map(M :: term()) -> bool().
```

returns `true` if M is a map otherwise `false`.

```
erlang:map_size(M :: map()) -> Size :: integer().
```

returns the number of key-value pairs in the map.
Same as, `length(maps:to_list(M))`.

functions

Converting to and from a list

```
maps:to_list(M :: map()) -> [{K1,V1}, ..., {Kn,Vn}].
```

Where the pairs, [{K1,V1}, ..., {Kn,Vn}], are returned in sorted order.

```
maps:from_list([{K1,V1}, ..., {Kn,Vn}]) -> M :: map().
```

Build a map from a list of key-value pairs.

Even more functions

`maps:new() -> M :: map().`

Returns a new empty map. Same as, `maps:from_list([])` or #{}

`maps:is_key(K :: term(), M :: map()) -> bool().`

Returns `true` if map `M` contains key `K`, otherwise it returns `false`.

`maps:get(K :: term(), M :: map()) -> V :: term().`

Returns the value `V` associated with key `K` if map `M` contains key `K`.
If no value is associated with key `K` then the call will fail with an exception

`maps:put(K :: term(), V :: term(), M0 :: map()) -> M1 :: map().`

Associates key `K` with value `V` and inserts the pair into map `M0`. If key `K` already exists, the old associated value is replaced by value `V`.

Functions continues

```
maps:find(K :: term(), M :: map()) ->
                                {ok, V :: term()} |error.
```

Returns a tuple `{ok, V}` with value `V` associated with key `K` if map `M` contains key `K`. If no value is associated with key `K` then the function will return `error`.

Functions

Deleting Keys

```
maps:delete(K0 :: term(), M0 :: map()) -> M1 :: map().
```

Removes the key `K0`, if it exists, and its associated value from map `M0` and returns a new map `M1` without key `K0`.

```
maps:without([K1, .., Kn] = Ks, M0 :: map()) -> M1 :: map
().`
```

Removes keys `K1` through `Kn`, and their associated values, from map `M0` and returns a new map `M1`.

Functions

```
maps:keys(M :: map()) -> [K1, ..., Kn].
```

Returns a complete list of Keys, in sorted order, which resides within map `M`.

Same as, `[K || {K,_} <- maps:to_list(M)]`.

```
maps:fold(F :: function(), I :: term(), M :: map()) ->
Result :: term().
```

Same as, `lists:foldl(fun({K,V}, Acc) -> F(K,V,Acc) end, I, maps:to_list(M))`.

Functions

```
maps:map(F :: function(), M0 :: map()) -> M1 :: map
().
```

Produces a new map `M1` by calling the function fun `F(K, V)` for every key `K` to value `V` association in map `M0` in defined order.

```
maps:merge(M0 :: map(), M1 :: map()) -> M2 :: map().
```

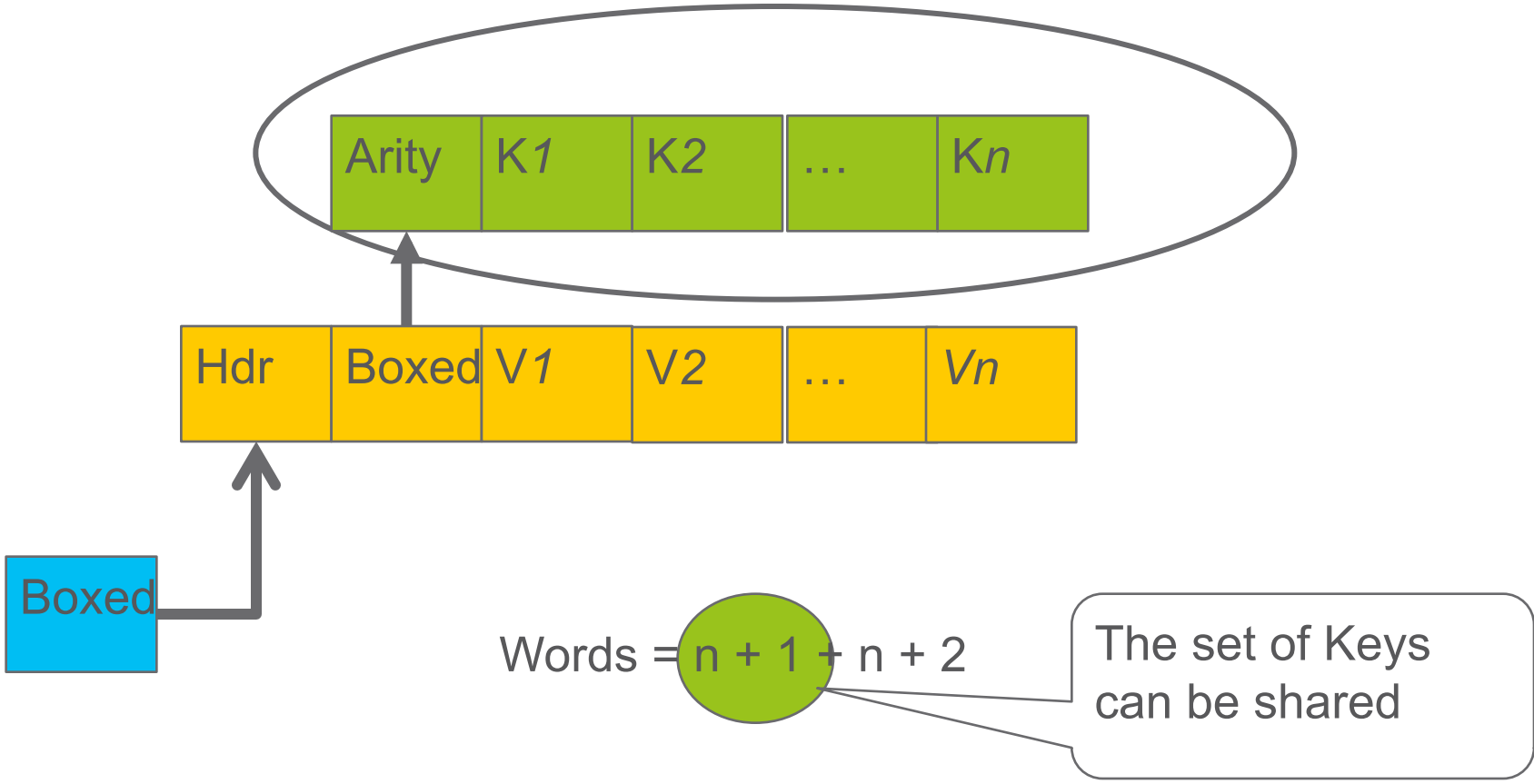
Merges two maps into a single map. If two keys are equal in both maps the value in map `M0` would be superseded by the value in map `M1`.

```
Ma = #{ a => 1, b => 2},
```

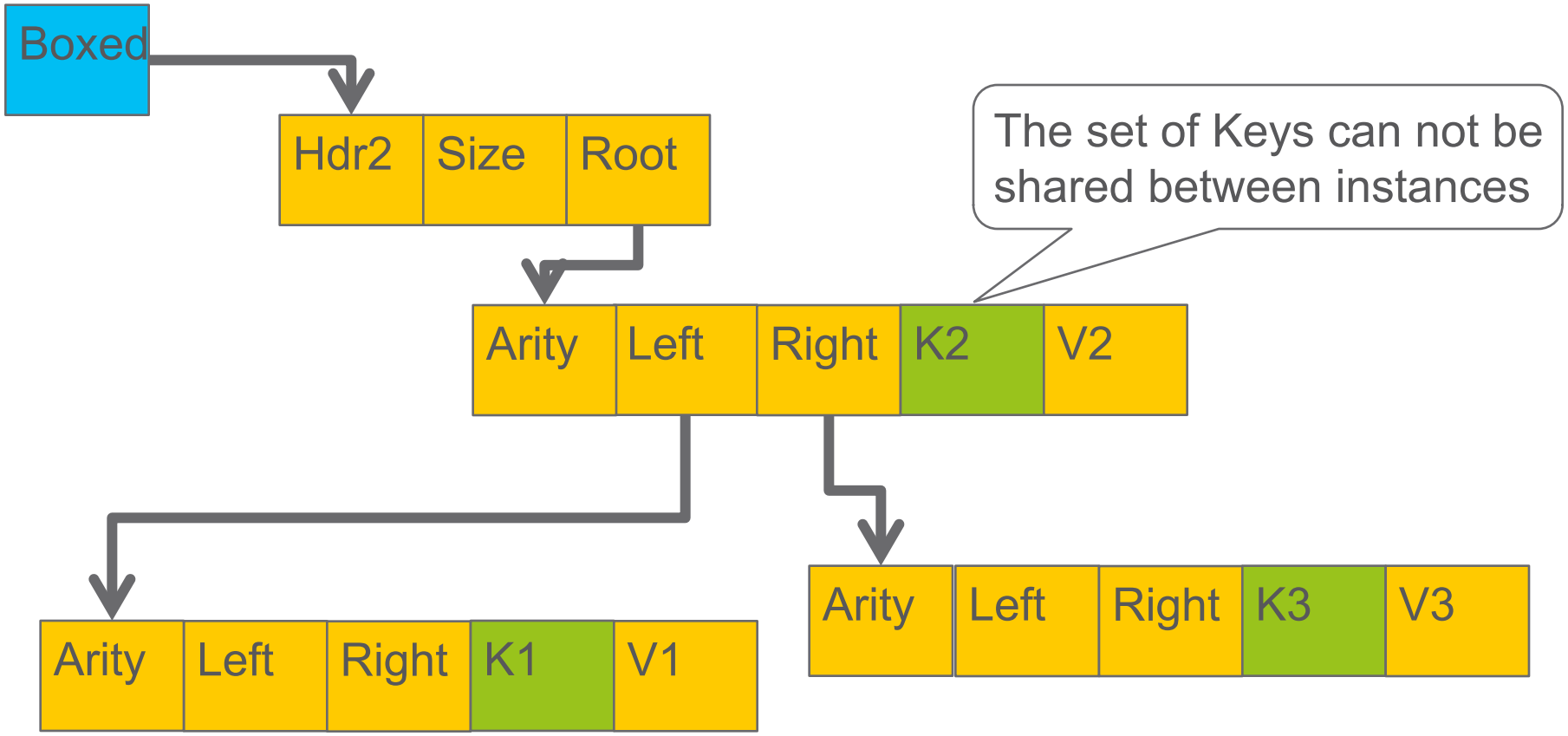
```
Mx = #{x => 14, y => 27}
```

```
2> M1 = maps:map(fun(K,V) -> V+10 end, Ma),
#{a => 11, b => 12}
3> M2 = maps:merge(Ma, Mx).
#{a => 11, b => 12, x => 14, y => 27},
```

Internal Representation 1

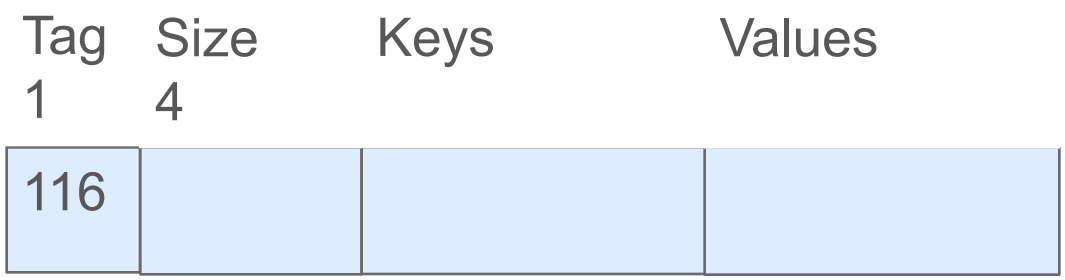


Internal Representation 2



Words = 3 + n*5

External representation



Use cases

- XML parser
- Json mapping
- ASN.1 mapping
- Replace most uses of property lists
- Used in API's instead of records
- Using a map for holding the state of a process

JSON MAPPING

Mochiwebs mochijson decodes Json dictionaries as the following:

```
{"key": "value"} -> {struct, [{"key", "value"}]}
```

This could instead be:

```
{"key": "value"} -> #{ "key" => "value"}
```

mOre json

```
{
   "glossary": {
      "title": "example glossary",
      "GlossDiv": {
         "title": "S",
         "GlossList": {
            "GlossEntry": {
               "ID": "SGML",
               "SortAs": "SGML",
               "GlossTerm": "Standard … Markup…",
               …,
              "GlossDef": {
                   "para": "A meta-markup
language, ...",
                 "GlossSeeAlso": ["GML", "XML"]
               },
               "GlossSee": "markup"
            }
         ...
```

```
#{
   "glossary" => #{
      "title" => "example glossary",
      "GlossDiv" => #{
         "title" => "S",
         "GlossList" => #{
            "GlossEntry" => #{
               "ID" => "SGML",
               "SortAs" => "SGML",
               "GlossTerm" => "Standa … Markup…",
               …,
               "GlossDef" => #{
                     "para" => "A meta-markup lan….",
                     "GlossSeeAlso" => ["GML","XML"]
                  },
                  "GlossSee" => "markup"
               }
            ...
```

More JSON continued

Lets find the value for **ID**

```
Decoded = json:decode(Json),
    #{ "glossary" => Glossary } = Decoded,
    #{ "GlossDiv" => GlossDiv } = Glossary,,
    #{ "GlossList" => GlossList }   = GlossDiv,
    #{ "GlossEntry" => GlossEntry } = GlossList,
    #{ "ID" => Id } = GlossEntry.
```

Or with single value access

```
Id = Decoded#{ "glossary" }#{ "GlossDiv" }#
{ "GlossList" }#{ "GlossEntry" }#{ "ID" }.
```

Used in API

- Replace records and get rid of include files in certain APIs

Example:

- `file:read_file_info(File)`

- Functions returning property lists could/should return maps instead
- `process_info(Pid)`

Open Questions

- Syntax for single value access?
  - is it necessary?
  - syntax?
- When to change internal representation from linear to tree? 32, 64?
- Should there be different syntax (or only a function) to add a new key vs. updating a key?
- Syntax for type specifications (and what can Dialyzer do about it?)

Summary

- **Working assumption**
  - Maps are ORDERED
  - Can have ANY TERM as KEY
  - Pattern Matching
  - Map comprehensions
- There are open questions
- A more detailed description will be published within a few days
- Your feedback is welcome
- The plan is to have Maps in R17B

ERICSSON