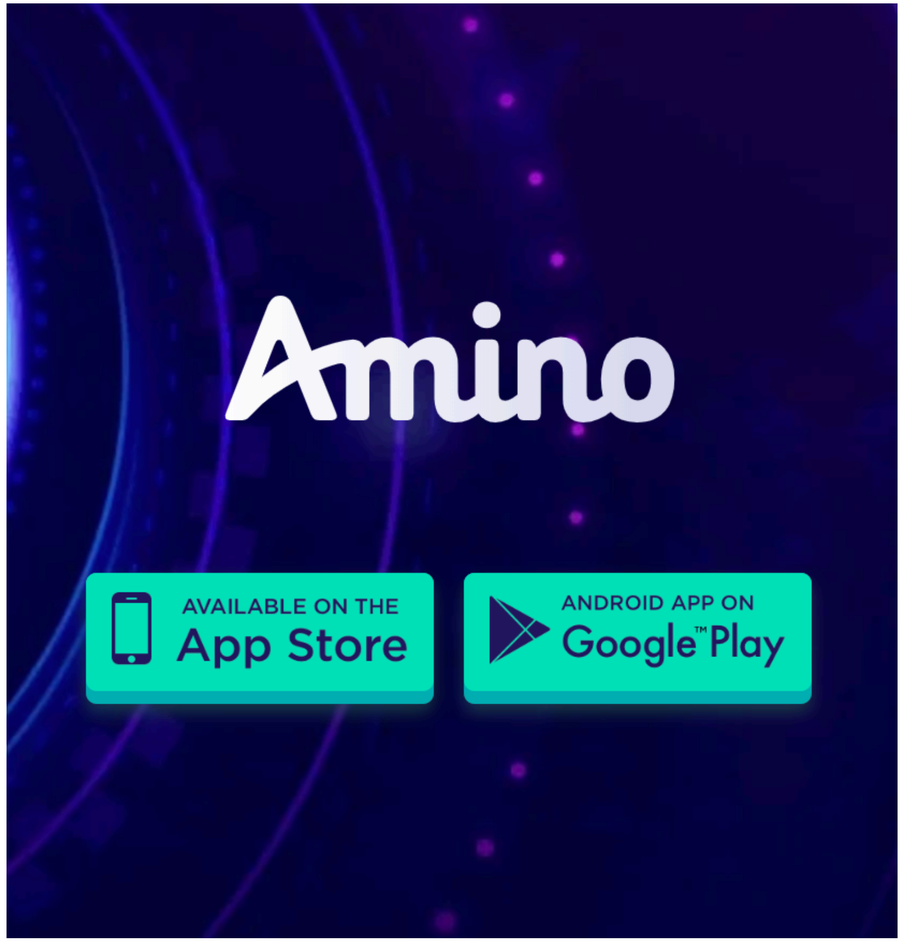
Bootstrapping a New Language with Ruby

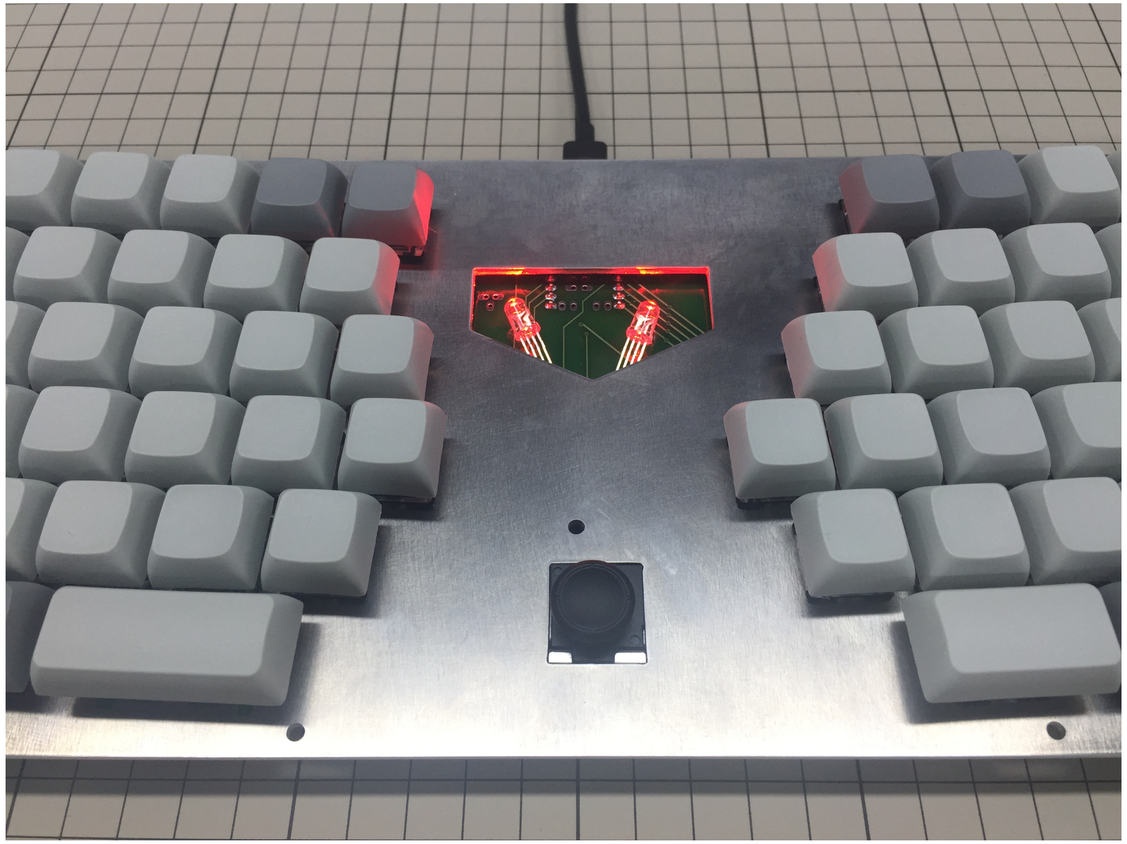
Me

- A Programmer
- @luikore in GitHub
- I work on Amino Apps (SH & NY)
  - Millions of communities: Anime, K-Pop, Sports, Games ...



Me

- Wheel re-inventor
- v2 keyboard -->
- v3 is on the way: analog input by eddy current detection, bit masking debounce algorithm, super fast response time...

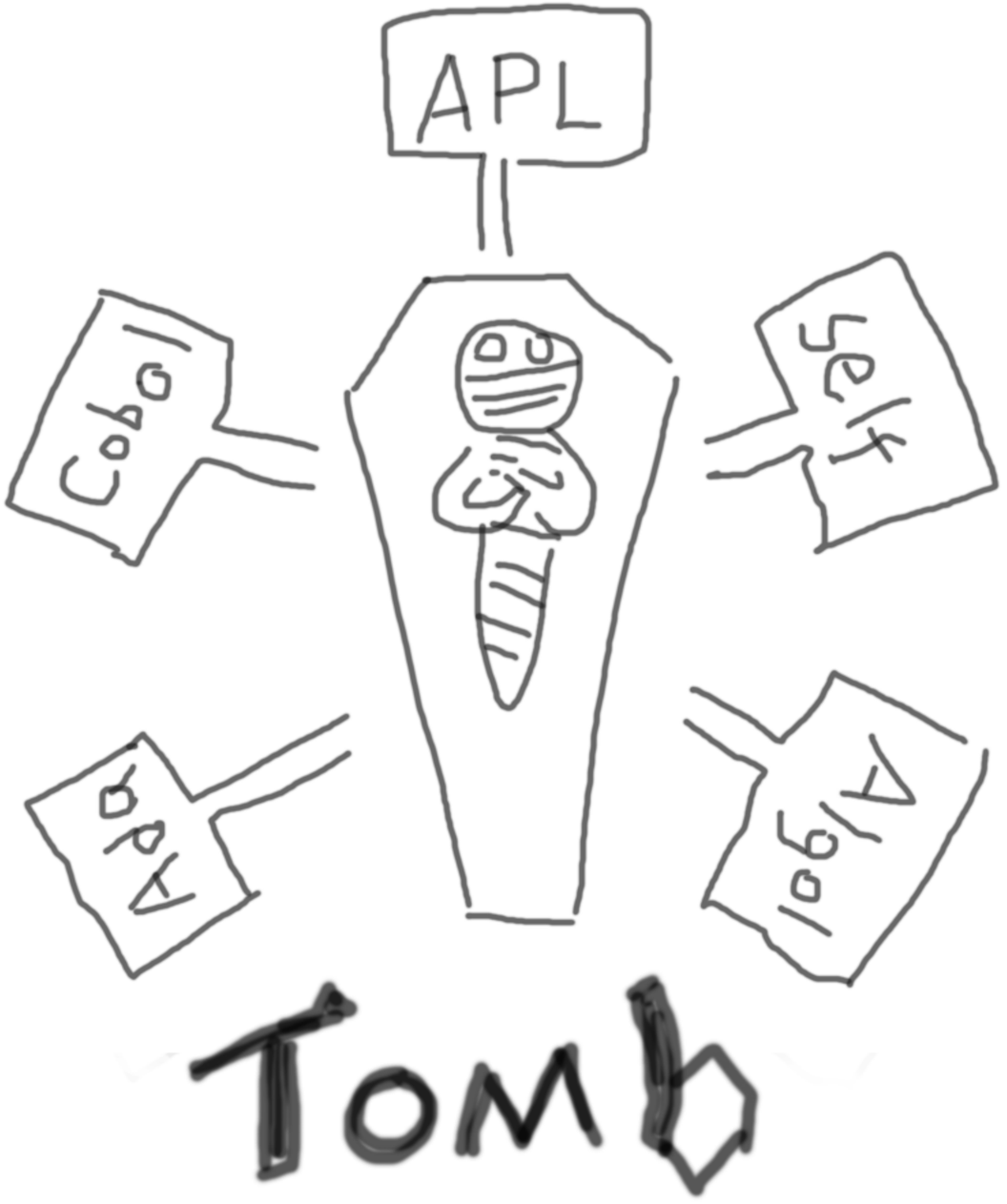


Disclaimer

- This is not yet another compiler course
- Just sharing some tiny experience of ~~life~~ wheel reinventing.

Programming Language

- Some people's idea: Ruby is dead.
- My idea: let's build a dead language from the beginning.


APL
Cobol
Self
Ada
Algol
Tomb

Overview

- **No VM** -- container is the most popular "VM".
- **No GC** -- Values are immutable, only scopes are mutable.
- **Language Oriented** -- Static check embed languages, and import libraries of other languages.
- **Functional** -- Syntax from Potion, Crystal, and Haskell
- **Gradual Typing** -- Typing declaration is optional, can add type checks later.

Syntax Design

- File extension: .to

Syntax: Juxtaposition

```
struct Bot[name :: String]

class Bot
  def hi
    puts (@greet "hi, I am " @name)
  end

  def greet prefix name
    prefix + name
  end
end

Bot["Ada"].hi
```

Syntax: Comma

```
[1, 2, 3]
# which can also be written as
[
  1
  2
  3
]
```

Syntax: Newline

```
buy "tomato" 3
if cheap? "melon"
  buy "tomato" 1
end
# which can also be written as
buy "tomato" 3
if cheap? "melon", buy "tomato" 1;
```

Syntax: Monadic

```
[1, 2, 3] -> x
  puts x
end

x <- [1, 2, 3]
puts x
```

Syntax: Sigils

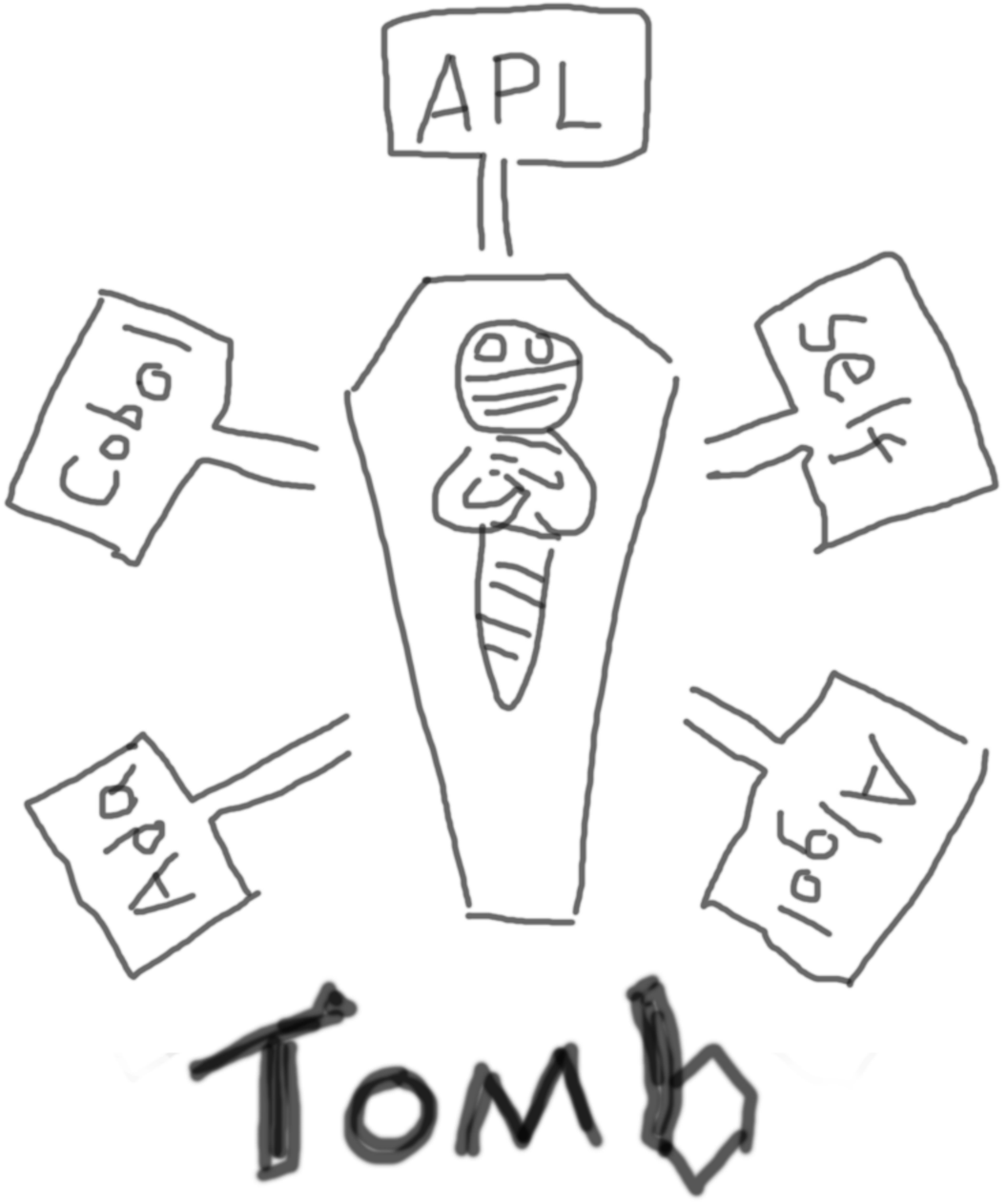
```
$r/b(a+)d/ =~ "baaad"
puts $1.size

$|parse|
  S: '(' S* ')'
  S: 'a'

$file

$sql"select * from eroge"
```

On Sigils

- A good way to avoid adding too many features -- learned from failure of Elixir
- Block sigil makes language composition clear -- learned from failure of Nemerle and Perl 6

Compiler

- To build a language, first we need a compiler
  - **Q**: What language should the compiler be written in?
  - **A**: The language we want to build.


APL
Cobol
Self
Ada
Algol
Tomb

In a Simple Word

```
./compiler compiler.to > compiler
```

Meta-Circular

- To break the meta-circular loop, we should start the compiler with some language...

Advantages of Ruby

- A compiler manipulates strings and Ruby regards strings as the most important data structure.

- The target language is similar to Ruby in many ways (every good new language tries to mimic Ruby).

Self Compiling

Replace the initial compiler with an interpreter in Ruby

```
ruby interpreter.rb compiler.to compiler.to > compiler
```

Parser Generator

- Parser is the first thing we need to do with a compiler
- Bison or ANTLR? The language will have its own parser generator so we can build new languages on top of it, without any external dependencies.
- So in additional to compiler.to, we also need a compiler_compiler.to, which parses compiler definitions -- yet another language we need to build, and generates compiler.to

- Simple format

```
.on
  BEGIN.lex
    (match $r"::") or (return false)

.lex
  \[
    token "op.array.begin"
  \]
    token "op.array.end"
  \{
    token "op.assoc.begin"
  \}
    token "op.assoc.end"
  \w+
    token "ty"
  =>
    token "op.return"
  \#[^\n]*$
    # ignore
  (?=[=\n])|\z
    yield_node (parse "Type")
    return true
  [\ \t]+
    # ignore
```

- Generated

```
$line'compiler/Expr.mb:55'
if match $r/(?:not|and|or)\b/
  token "op.logic.clause" (extract_string 0)
  next
end
$line'compiler/Expr.mb:57'
if match $r/\.\.\./
  token "op.range.exclusive"
  next
end
$line'compiler/Expr.mb:59'
if match $r/\.\./
  token "op.range.inclusive"
  next
end
$line'compiler/Expr.mb:62'
if match $r/[~!]/
  # must be after !=
  token "op.prefix" (extract_string 0)
  # TODO knot operators
  # NOTE: since we don't have our own regexp engine ye
  #       look-behind should not go back beyond the la
  #       it is impossible for onigmo to support ancho
  next
end
$line'compiler/Expr.mb:72'
if match $r/(?<!\s)\(/
  token "op.suffix.subscript"
  next
end
$line'compiler/Expr.mb:74'
if match $r/[\(\[\{]/
  token "op.bra" (extract_string 0)
  next
end
$line'compiler/Expr.mb:76'
```

Nested Word

- Lex directed, Not syntax directed -- it is incremental and can be reused in text highlighting and auto indenting.
- Nested Word (Visibly Pushdown) lexer -- composible
- Just use simplest LL(1) parser, the whole system is quite powerful -- I can translate any LL(*) into it.

Bootstrapping Commands

```
ruby interpreter.rb compiler_compiler.to compiler.mb > compiler.to
ruby interpreter.rb compiler.to compiler.to > compiler.c
```

Validate the Target Compiler

```
cc compiler.c -o compiler
./compiler compiler.to > compiler-2.c
diff compiler.c compiler-2.c
```

Should get the same result!

Programming before You Have the Language

- This is programmer's tradition.
- The first programmer, Ada Lovelace programs for the mechanical computer before it is built by Charles Babbage (tho he never finished the computer after many years of work).

What I Did First

- Create a TextMate grammar first.
- TextMate Grammar also uses Oniguruma -- the regular expression in Ruby, also the regular expression library our compiler will link to.
- So we can start testing regular expressions with editor.

Target Code Generating

Code Generating

- I am not compiler expert. At least you can release when compile to C.

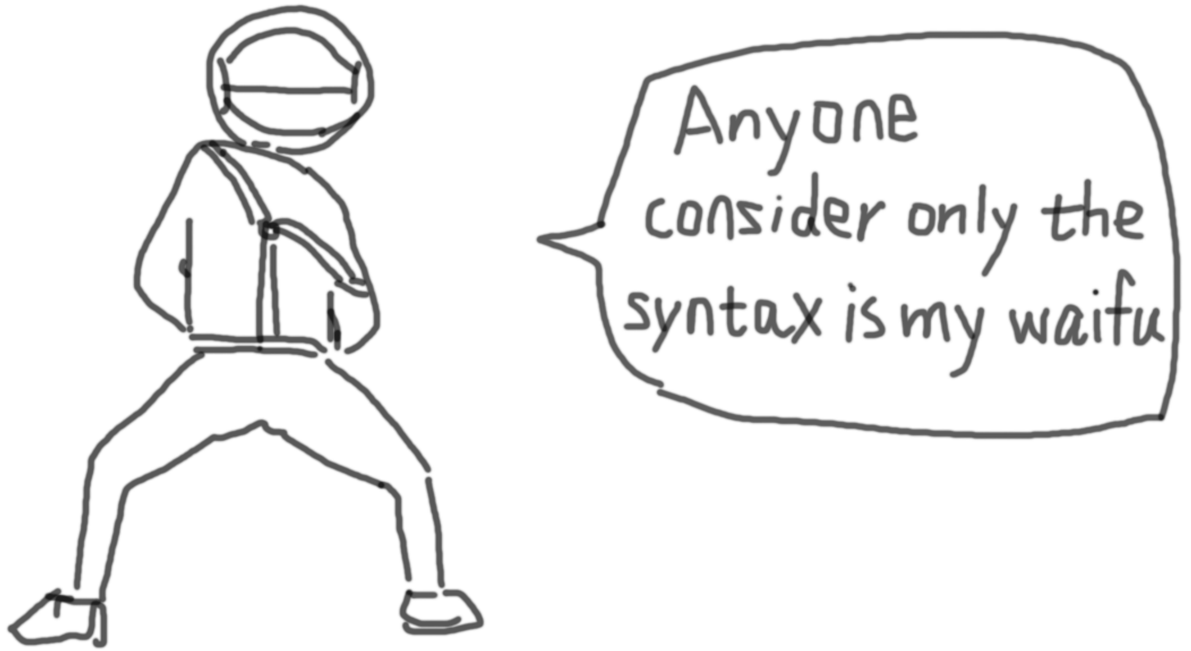
Clang & C18

- UTF-8 strings -- `u8"\u9527"`
- Thread local variables
- Lambdas (blocks)
- Structs and basic types

Clang & C18

- `_Static_assert` -- can implement more static type checks than the C type system.
- Auto insert destructing code at end of scope: `__attribute__((__cleanup__(func))) Type var = ...`
- Allows `$` sign in names -- so no conflict for names from source language and runtime. (vlang should use this)
- A lot of attributes in GCC.

Semantic & Runtime: A Lot More to Consider than Syntax



Value Representation

- Why we need?
- To make a better type system than C
- Allow dynamic dispatch

Fat Pointers

- Java uses 2 representations: native types and OOP (Object and boxed native types) -- **NOT** everything is an object.
- Ruby uses tagged pointer -- have to check if the object is a pointer or value every time we use, this is **SLOW**
- More and more languages use fat pointers: Rust and Go.

Fat Pointers

Lazy man's Choice

Fat Pointer: Basic Idea

```
struct String {
  int32_t klass;
  int32_t size;
  char* ptr;
};

void strlen(char* ptr) {
  int size = ((struct String*)(ptr - 1))->size;
  return size;
}

void f() {
  struct String s = {KLASS_STRING, 4, "jojo"};
  strlen(&(s->ptr));
}
```

Fat Pointers

- It uses a bit more space for native types, but can be optimized by compiler.
- C interface is simpler: pass the essential pointer instead of the wrapped pointer.
- No need boxing/unboxing values -- MJIT and Graal VM eliminates value convertions on some cases, but we don't need this optimization at all!

Memory Management

- A controllable, high performance GC is a huge beast.
- Reference counting is much simpler -- Ruby also uses ref-count to manage compiled regular expression literals.

Is Refcount Slow?

- Deferred Reference Counting: refcount a group of objects instead of individual objects.
- Some method doesn't change the refcount of a parameter, so the refcounts can be removed.
- Generational object allocation: in leaf generations, we can remove all the refcount code -- similar to transient heap in Ruby

Problem: Loop Reference

- This problem bothers programmers for a long time -- the old IE bug for example.
- Python uses a trick similar to tracing GC to solve this problem -- too complex!

Immutability: the Solution

- Immutability means you can't change an object after you created it.
- Also means no loop reference any more.
- So no need to solve the problem!

Immutability

Lazy man's Choice

Immutability:
Hard to Use?

- People (including me) finds it sometimes hard to write code with immutable objects.
- Lens and record-update syntax in Haskell helps it.
- We can implement Lens in Ruby-like syntax, which makes it even more easier!

Problem:
Immutable Arrays & Maps

- Another Problem: it copies a LOT when using immutable arrays and hash maps.

- One of the main performance problems which hurts the adoptions of Haskell.

Solution: HAMT & HATrie

- Use tree to represent big arrays and maps, so we only copy a small piece of memory when making a new one.
- They are wide hierarchical trees, each node is a sparse array. We can use a special instruction: popcnt to optimize sparse array.
- A Ruby Hash for 6 million wikipedia title costs several GB memory, while HATrie uses only 100M. (I also made a gem for that)

Solution:
Mutable Implementation

- Sometimes right after we build an immutable object, we put it back to the variable, and throw away the old one.
- This can be optimized to mutable object field updates.
- While in user's perspective, the object is still immutable!

Which Object Can be Made Mutable?

- Static analysis: escape analysis to find out object that can be freed at some point.
- Dynamic analysis: object with refcount=1 and the assigning variable is being freed (refcount for the win!).

```
# what we write:

point = Point[1, 1]
point.y = 2

# with lens translation:

point = Point[1, 1]
point = Point[point.x, 2]

# with single-refcount optimization:

point = Point[1, 1]
point.y = 2
```

Summary

- Language re-inventing is big topic, without simplification there is no way to finish the work.
- More todo: debugger integration, tracing, ...

Thanks