

Growing the Ruby Interpreter

Koichi Sasada

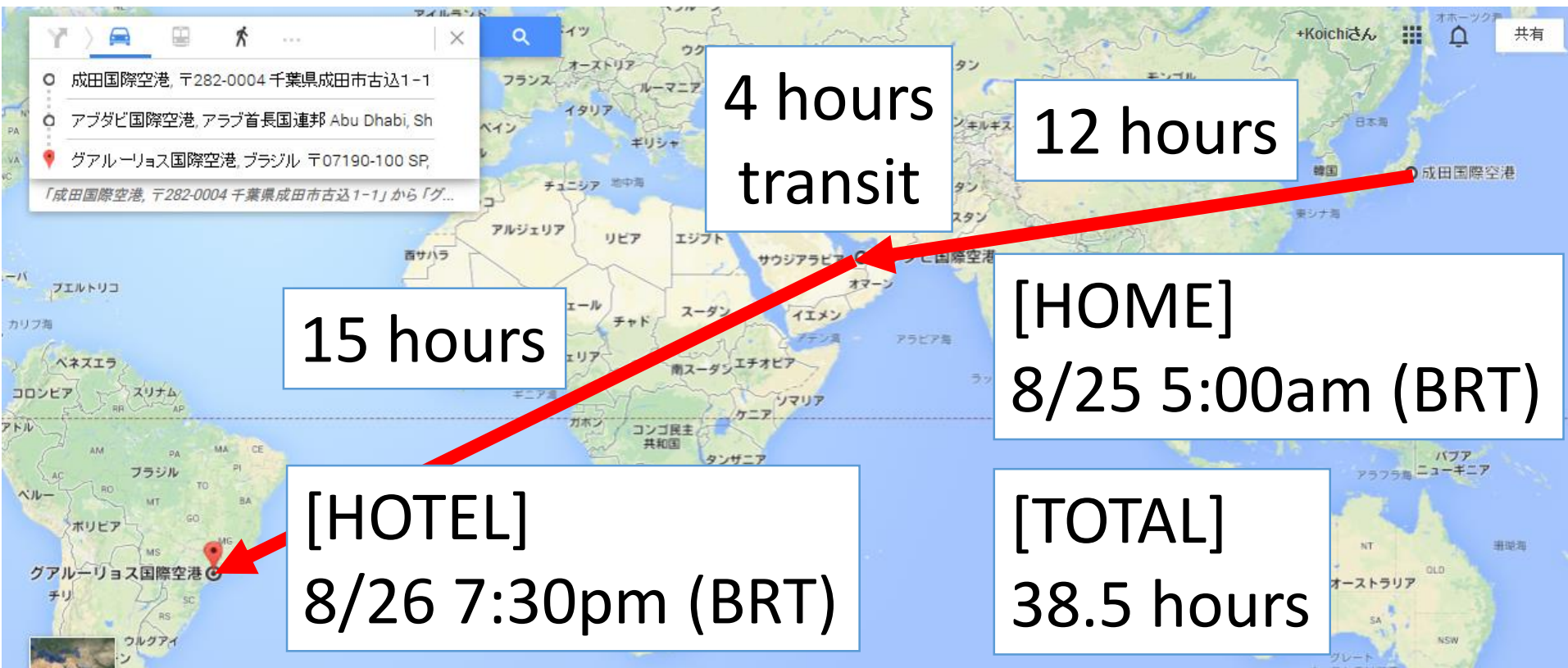
<ko1@heroku.com>



Today's talk

- Ruby 2.1 and Ruby 2.2
- How to grow up the Ruby interpreter?
 - Evaluator
 - Threading
 - Object management / Garbage collection

Koichi Sasada as a Japanese



Koichi Sasada as a Programmer

- CRuby/MRI committer
 - Core components developer
 - Virtual machine (YARV) since 2004/1/1
 - Rewrote Threads, GC, and so on
- Matz team at Heroku, Inc.
- Director of Ruby Association





Ruby Association

- Foundation to encourage Ruby dev. and communities
- Activities
 - Ruby programmer certification program
 - <http://www.ruby.or.jp/en/certification/examination/> in English
 - Grant project. Submit your proposal now!
 - 3 projects. About 5,000 USD per project (deadline: 3rd, Oct)
 - <http://www.ruby.or.jp/en/news/20140805.html>
 - Ruby Prize
 - Maintenance of Ruby (Cruby) interpreter
 - Events, especially RubyWorld Conference
 - **Donation** for Ruby developments and communities



Heroku, Inc. <http://www.heroku.com>

A screenshot of the Heroku website homepage. The background is a dark purple. At the top left is the Heroku logo. To its right is a navigation menu with links for 'Features', 'Pricing', 'Add-ons', 'Blog', 'Documentation', 'Support', and 'Contact'. Further right are 'Log in or' and a 'Sign up' button. The main content area features the headline 'Build, run, and scale apps.' followed by the subtext 'Cloud computing designed and built for developers.' Below this is a large white button with the text 'Sign up for free' and the smaller text 'No credit card required' underneath it.



Heroku, Inc. <http://www.heroku.com>

Ask Nando Vieira for more details



Growing the Ruby interpreter, Koichi Sasada,
RubyConf Brasil 2014



- Heroku supports OSSs
 - Many talents for Ruby, and also other languages
 - Heroku employs 3 **Ruby interpreter core developers**
 - Matz
 - Nobu
 - Ko1 (me)
 - We name our group “Matz team”

Matz

Famous title collector

- He has so many (job) title
 - Chairman - Ruby Association
 - Fellow - NaCl
 - Chief architect, Ruby - Heroku
 - Research institute fellow – Rakuten
 - Chairman – NPO mruby Forum
 - Senior researcher – Kadokawa Ascii Research Lab
 - Visiting professor – Shimane University
 - Honorable citizen (living) – Matsue city
 - Honorable member – Nihon Ruby no Kai
 - ...
- This margin is too narrow to contain



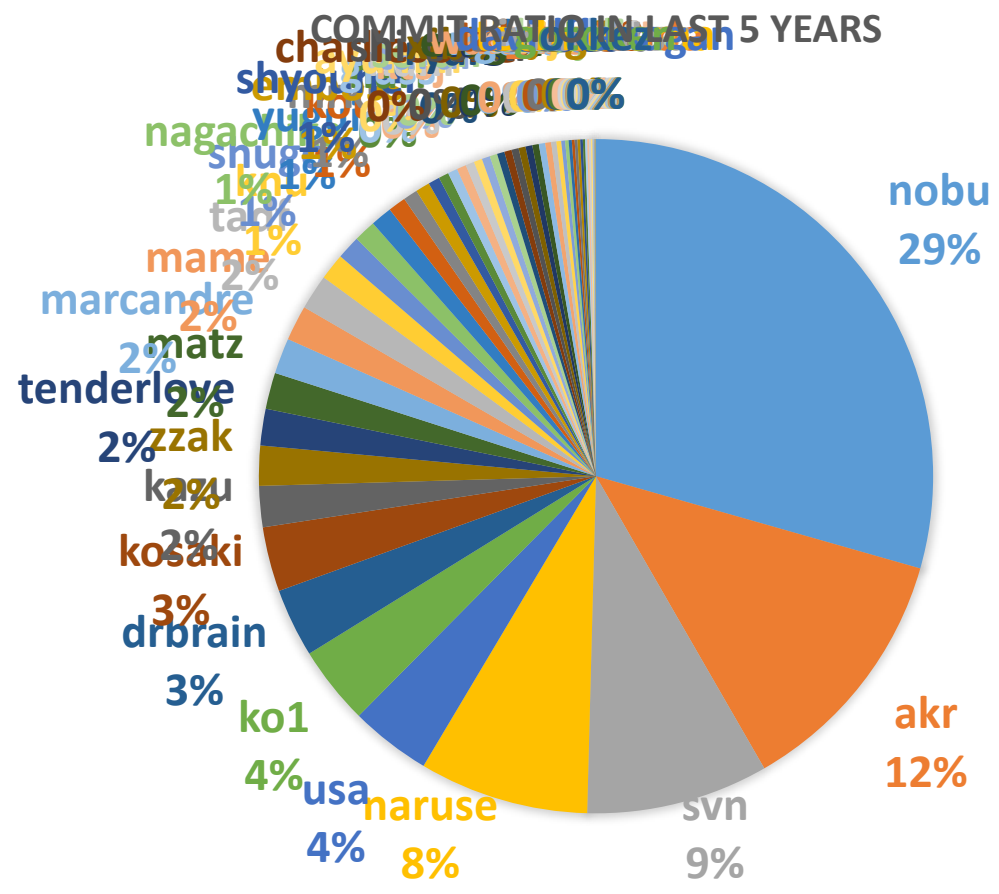
Nobu

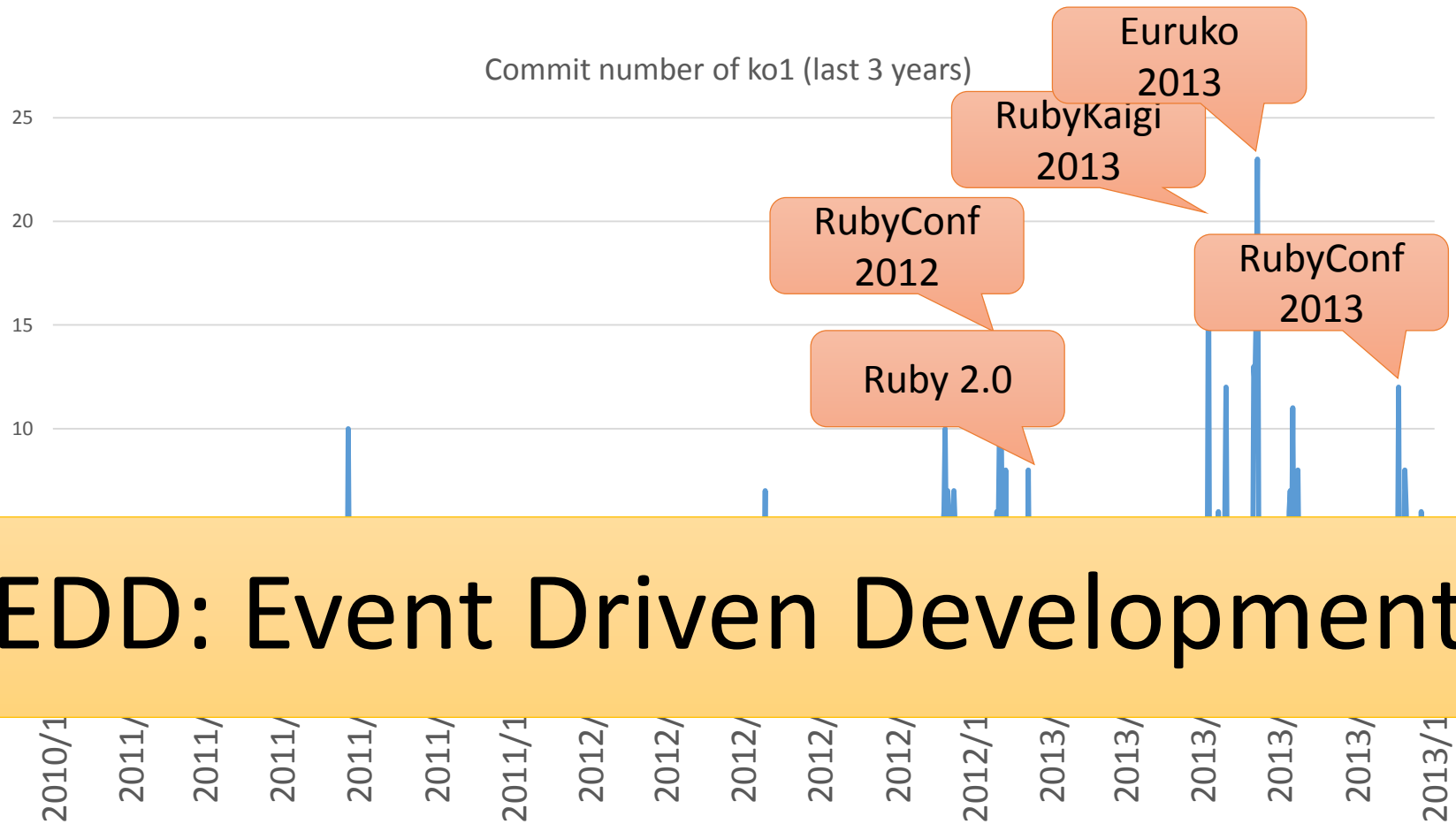
Patch monster

- Great patch creator



Nobu is Great Patch Monster





EDD: Event Driven Development

“Mission of Matz team”

**Improve quality of
next version of CRuby**

“Mission of Matz team”

- **Improve quality of next version of CRuby**
 - Matz decides a spec finally
 - Nobu fixed huge number of bugs
 - Ko1 improves the performance
- Next version of CRuby is “Ruby 2.2.0”

Ruby 2.1

Current stable



<http://www.flickr.com/photos/loginesta/5266114104>

Ruby 2.1

a bit old Ruby

- **Ruby 2.1.0** was released at **2013/12/25**
 - New features
 - Performance improvements
- **Ruby 2.1.1** was released at 2014/02/24
 - Includes many bug fixes found after 2.1.0 release
 - Introduce a new GC tuning parameter to change generational GC behavior (introduce it later)
- **Ruby 2.1.2** was released at **2014/05/09**
 - Solves critical bugs (OpenSSL and so on)

Ruby 2.1 New syntax

- New syntaxes
 - Required keyword parameter
 - Rational number literal
 - Complex number literal
 - `def` returns symbol of method name



<http://www.flickr.com/photos/rooreynolds/4133549889>

Ruby 2.1 Runtime new features

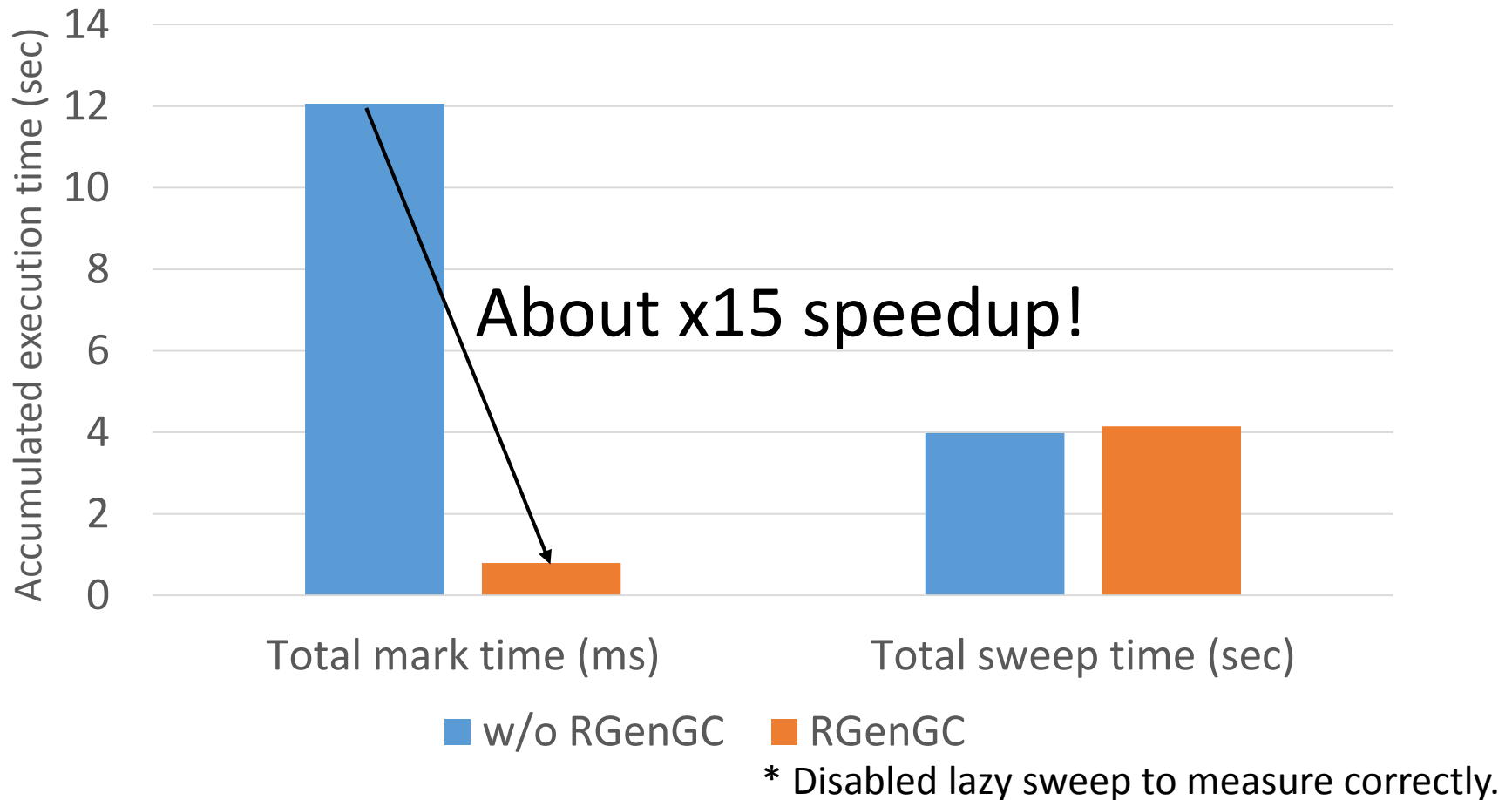
- `String#scrub`
- `Process.clock_gettime`
- `Binding#local_variable_get/set`
- Bignum now uses GMP (if available)
- Extending ObjectSpace

Performance improvements

- Optimize “string literal”.freeze
- Sophisticated inline method cache
- Introducing Generational GC: RGenGC

RGenGC

Performance evaluation (RDoc)

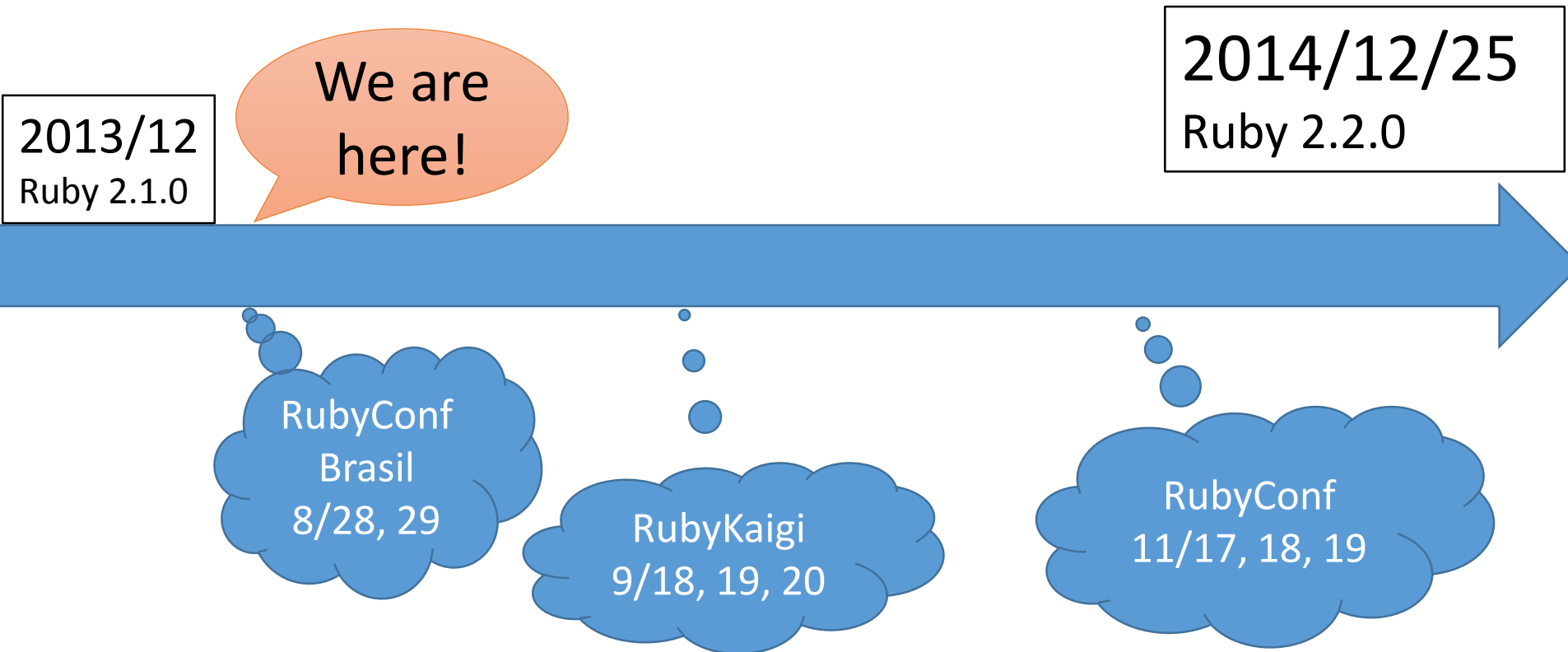




<http://www.flickr.com/photos/adafruit/8483990604>

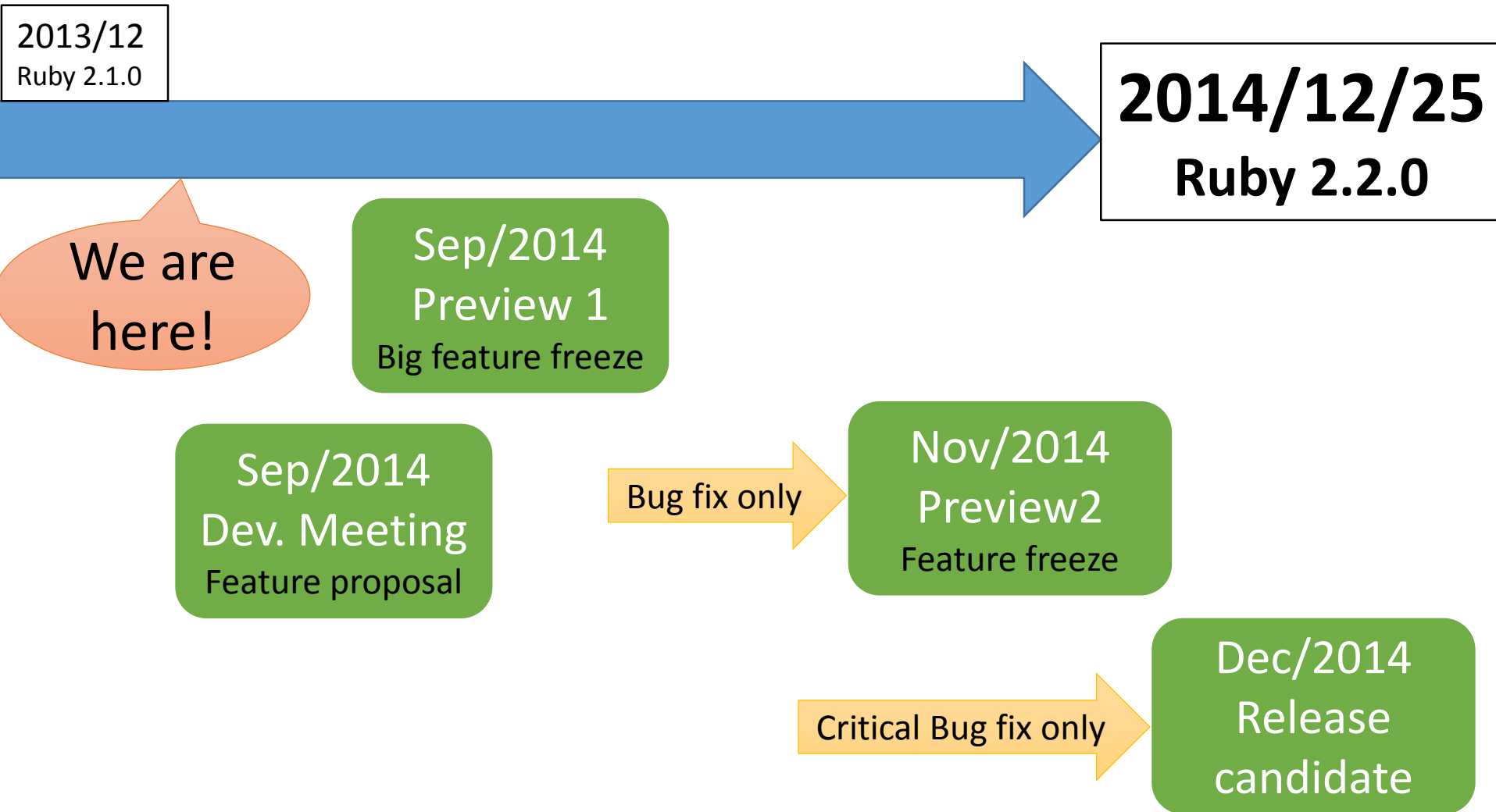
Ruby 2.2 Next version

Ruby 2.2 schedule



**Events are important for
EDD (Event Driven Development) Developers**

Ruby 2.2 (rough) schedule



2.2 big features (planned)

- New syntax: not available now
- New method: no notable methods available now
- Libraries:
 - Minitest and test/unit will be removed (provided by bundled gem)

2.2 internal changes

- Internal
 - C APIs
 - Hide internal structures for Hash, Struct and so on
 - Remove obsolete APIs
 - GC
 - Symbol GC (merged recently)
 - More ages strategy to reduce too-fast-promotion
 - Incremental GC to reduce major GC pause time
 - VM
 - More sophisticated method cache

Ruby 2.2 internals

Symbol GC

```
1_000_000.times{|i| i.to_s.to_sym}  
p Symbol.all_symbols.size
```

```
# Ruby 2.1
```

```
#=> 1,002,376
```

```
# Ruby 2.2 (dev)
```

```
#=> 25,412
```



<http://www.flickr.com/photos/donkeyhotey/8422065722>

NOTE: Drink a drop of water

Growing up the Ruby Interpreter

How do we grow up the Ruby interpreter?

Software consists of
many components

Ruby's components for users

Ruby (Rails) app

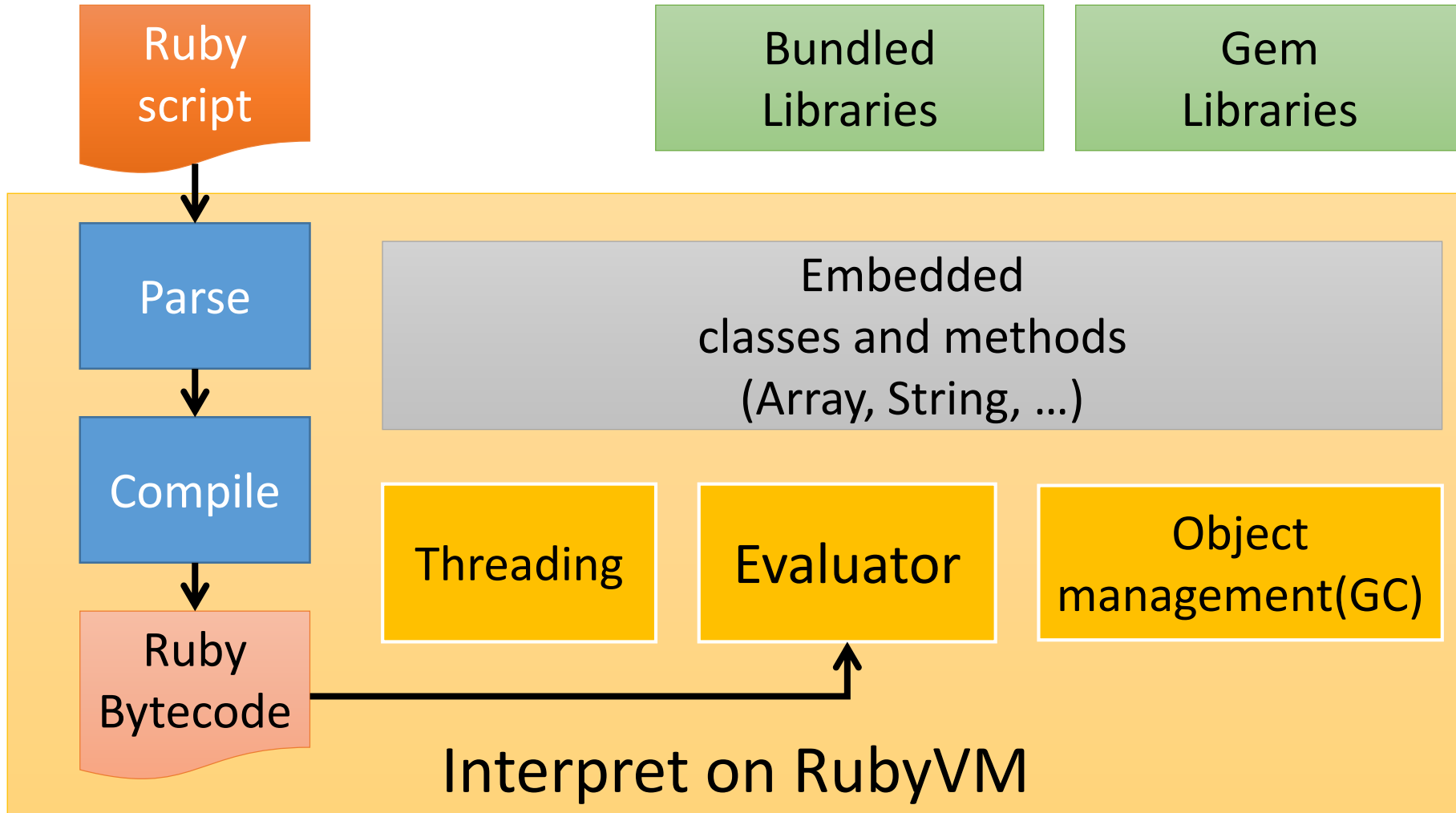
i gigantum umeris insidentes
Standing on the shoulders of giants

So many gems
such as Ruby on Rails (ActiveSupport, ...) and so on.

RubyGems/Bundler

Ruby interpreter

Ruby's components from core developer's perspective



My contributions

- Improve the performance for
 - Evaluator (10 years)
 - Thread management (10 years)
 - Memory management (recent years)

History of Ruby interpreter

1993 2/24
Birth of Ruby
(in Matz' computer)

1996/12
Ruby 1.0

1999/12
Ruby 1.4

2003/8
Ruby 1.8

2013/2
Ruby 2.0

1995/12
Ruby 0.95
1st release

1998/12
Ruby 1.2

2000/6
Ruby 1.6

2009/1
Ruby 1.9.0

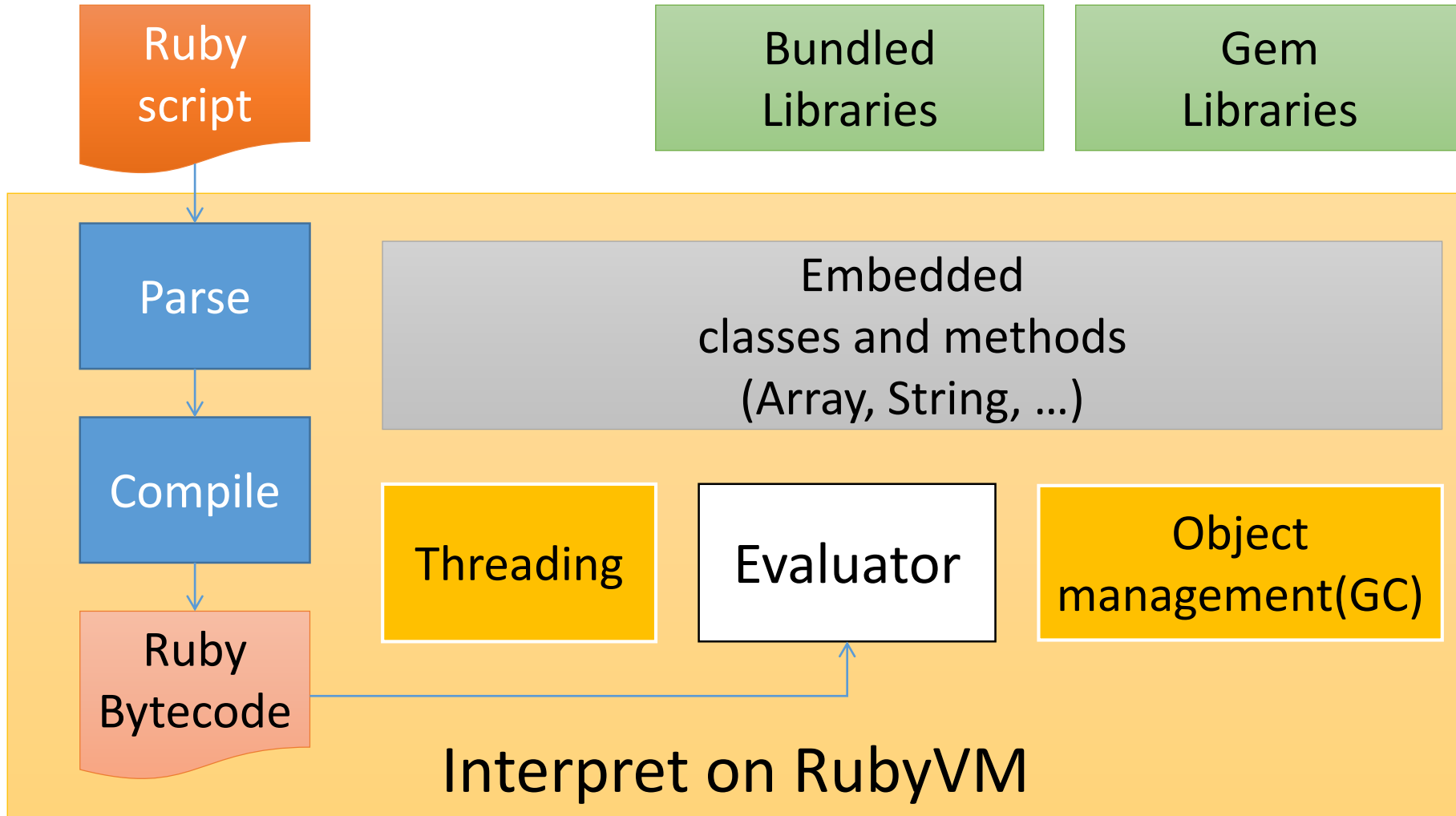
2013/12
Ruby 2.1.0

2004/1
YARV development

2013/3
RGenGC

Grow up Ruby interpreter by modification of core components

Evaluator

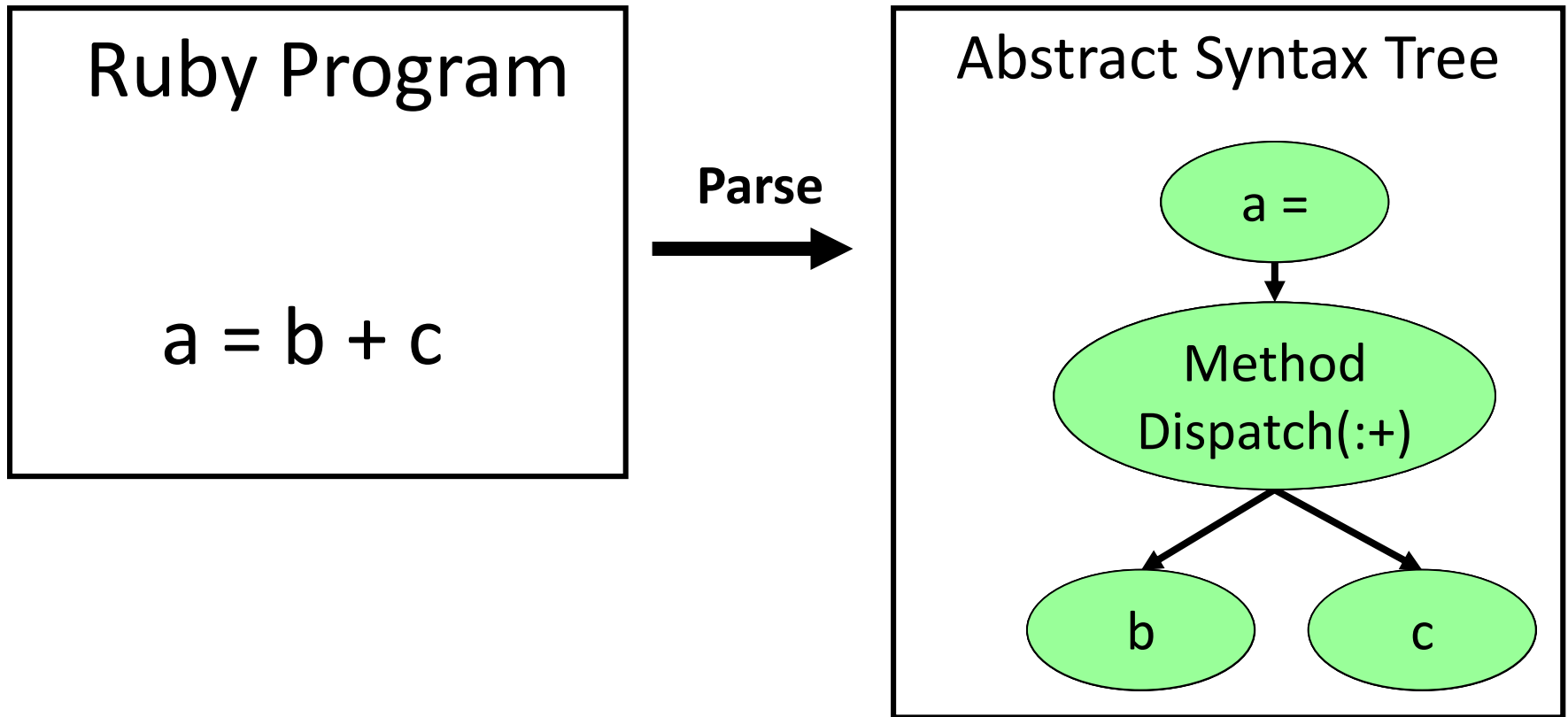


Evaluator

- Named YARV: Yet another RubyVM
 - Start until 10 years ago (2004/01/01)
 - Simple stack machine architecture
 - Execute each bytecode instructions one by one
- Apply many known optimization techniques

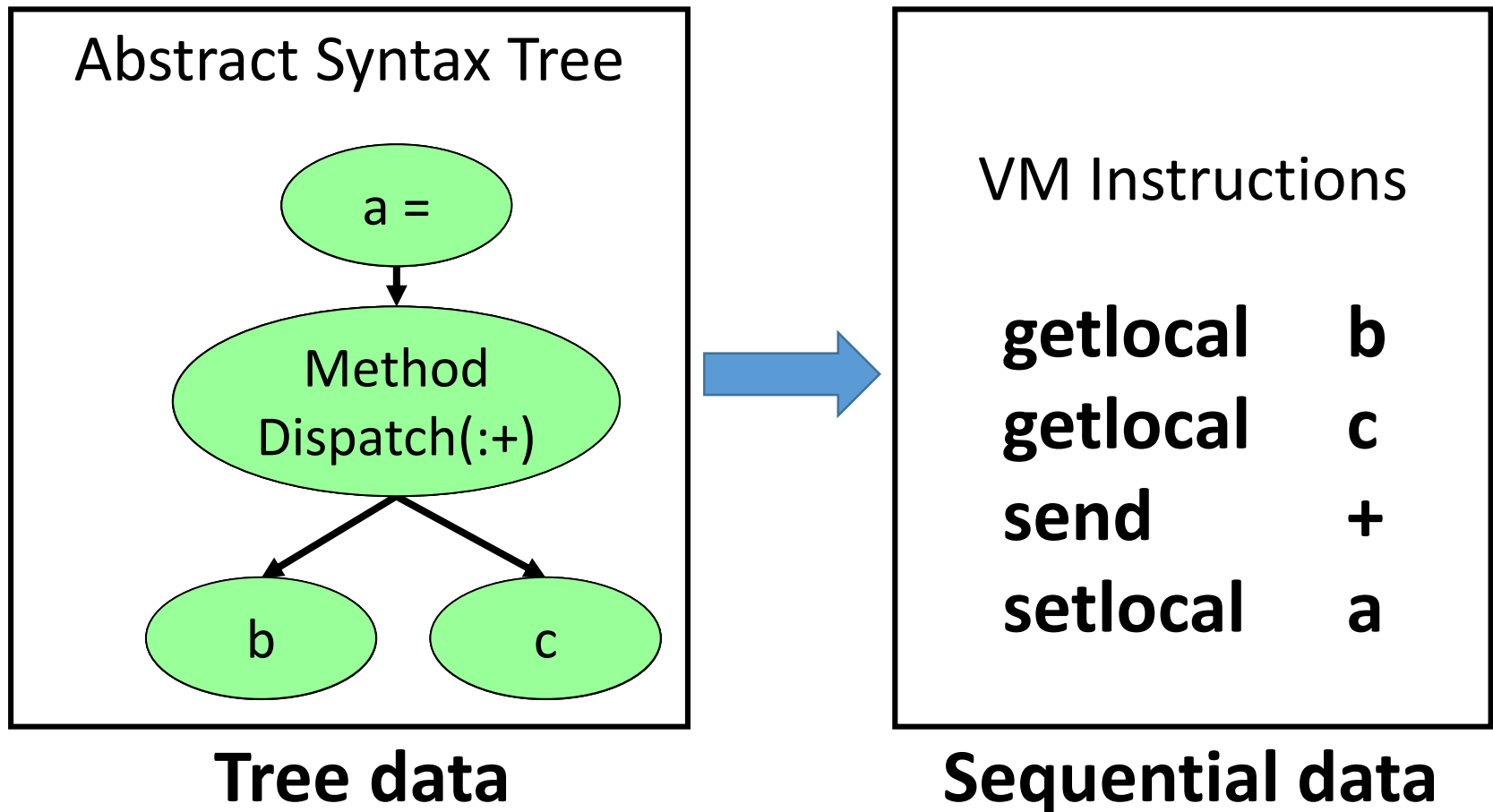
Evaluator

Compile Ruby to AST



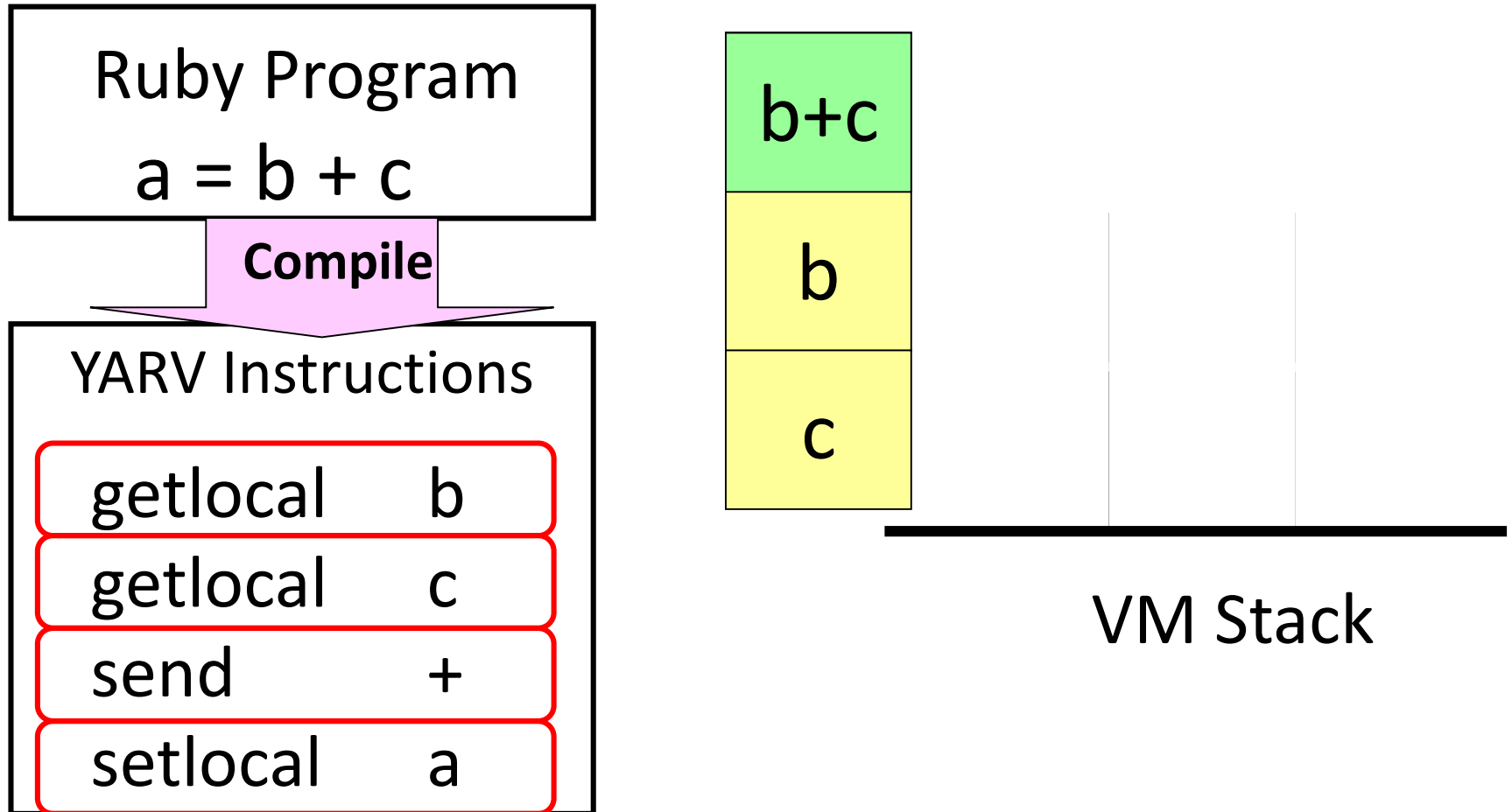
Evaluator

Compile AST to Bytecode



Evaluator

Execution as stack machine



Evaluator Optimizations

- Apply many techniques to improve performance
 - Peephole optimizations
 - Specialized instructions
 - Stack frame layout
 - Efficient exception handling
 - Efficient block representation
 - Direct threading
 - Stack caching
 - Instructions and operands unifications
 - ...

21

22

23

24

25

26

27

28

31

32

33

34

35

36

37

38

41

42

43

44

45

46

47

48

51

52

53

54

55

56

57

58

61

62

63

64

65

66

67

68

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

838

839

840

841

842

843

844

845

846

847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

912

913

914

915

916

917

918

919

920

921

922

923

924

925

926

927

928

929

930

931

932

933

934

935

936

937

938

939

940

941

942

943

944

945

946

947

948

949

950

951

952

953

954

955

956

957

958

959

960

961

962

963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

991

992

993

994

995

996

997

998

999

1000

Evaluator

Optimizations: Basic concept

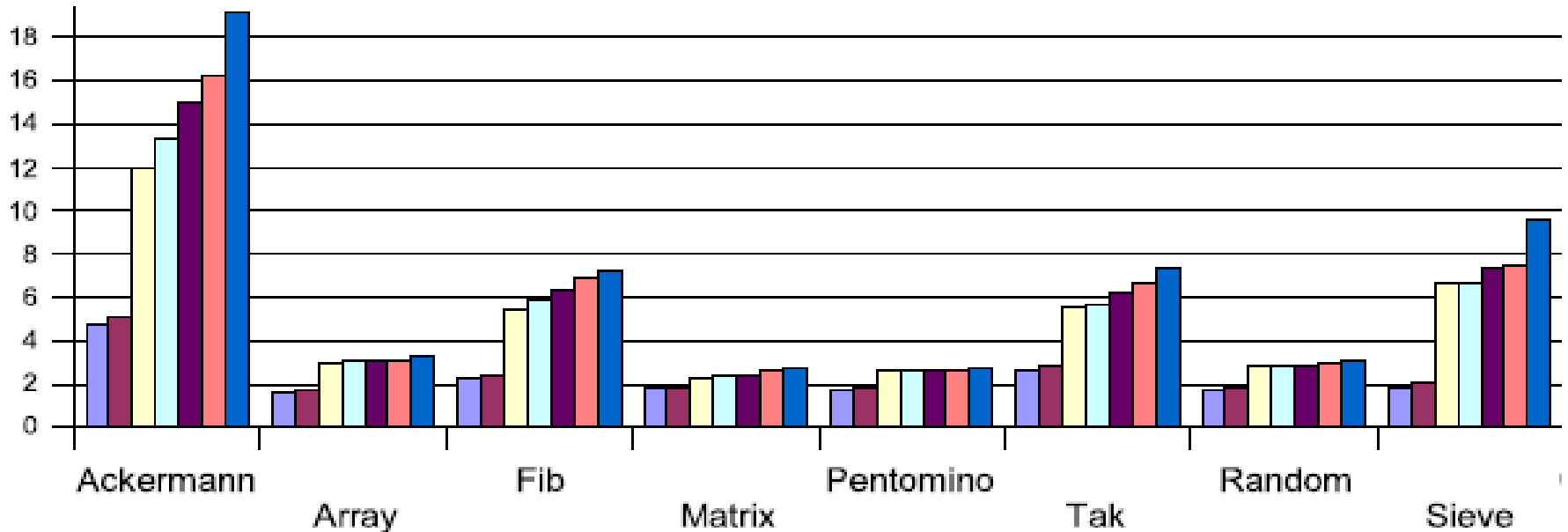
- Analysis usage

And optimize for frequent cases

- Example: Exception handling
 - Exceptions occur `*EXCEPTIONAL*` so optimize for no-exception control flow

Performance evaluation compare with Ruby 1.8

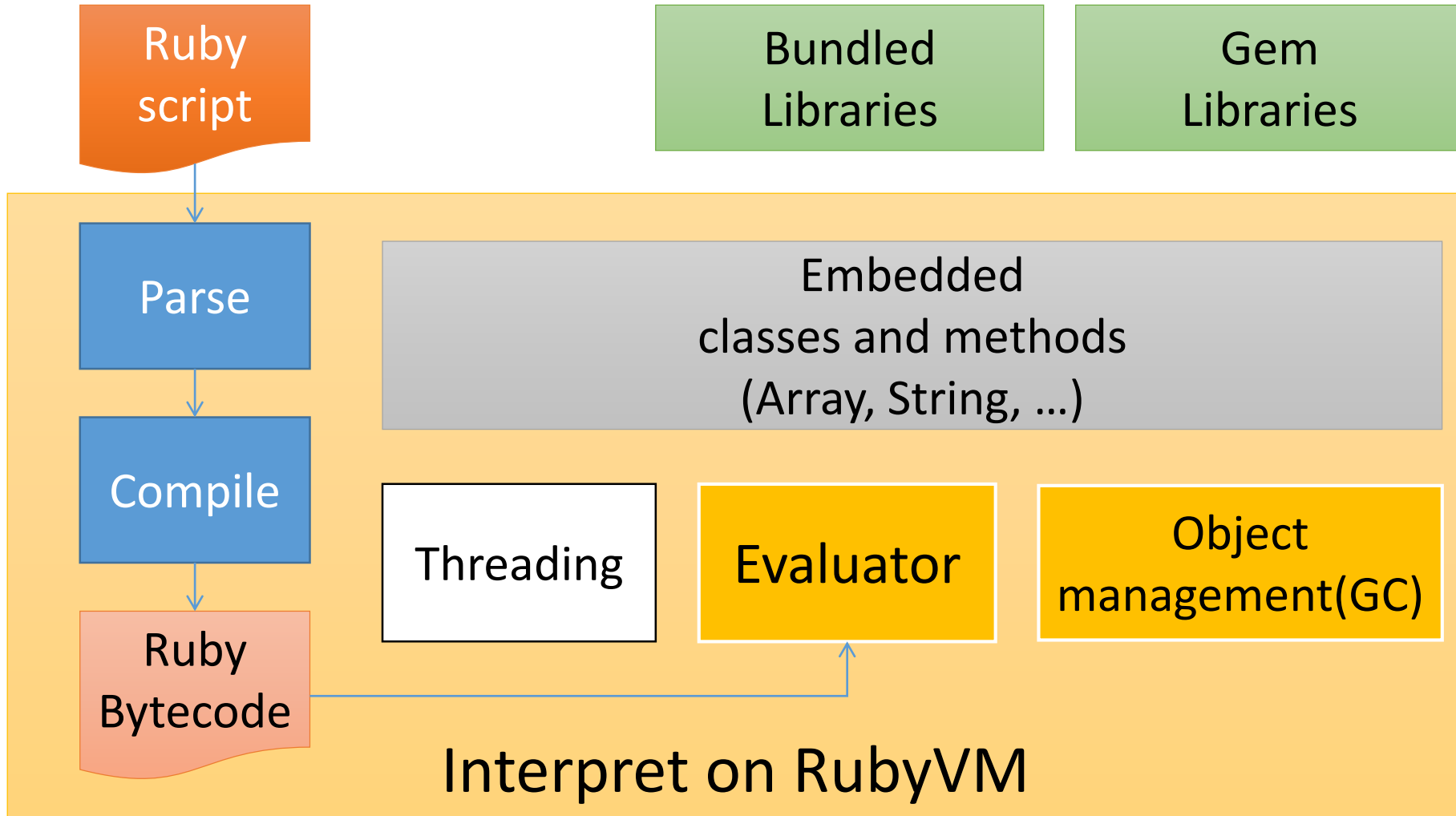
Higher is good



Main components

- Evaluator
- Thread management
- Memory management

Threading



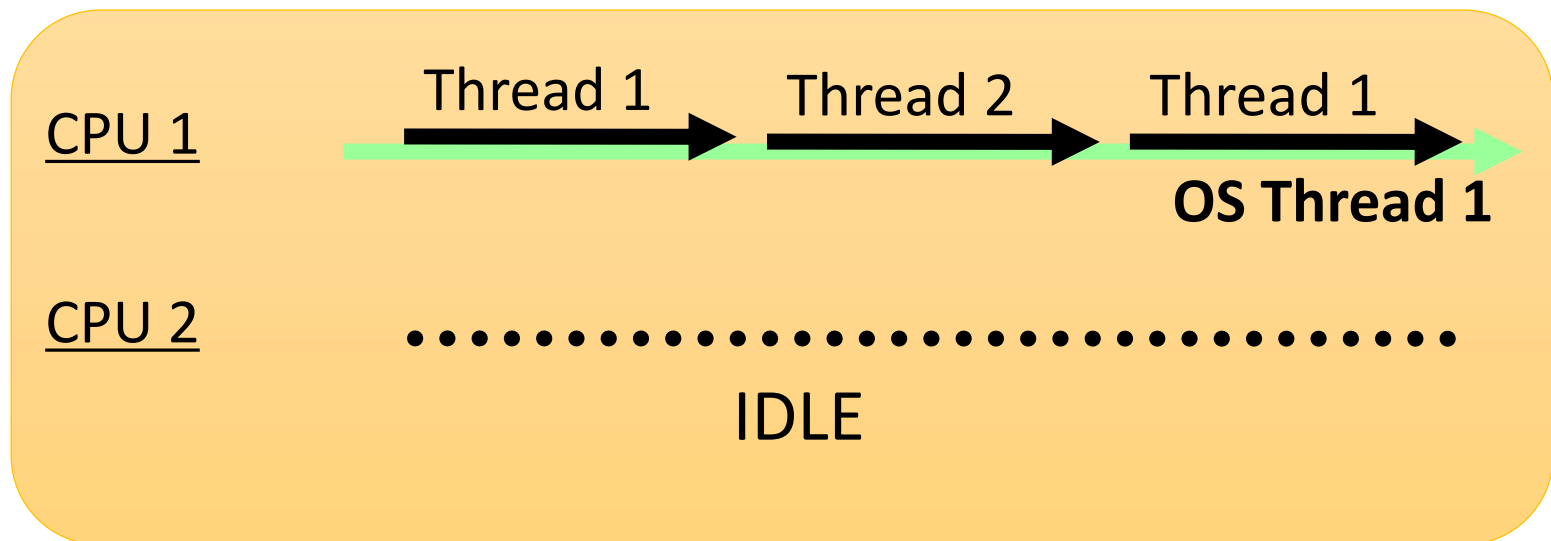
Threading

- Using native threads for each Ruby threads
- Parallel ruby execution is prohibited by GVL
 - You can free GVL if you write a code carefully in C level and run it in parallel

Threading

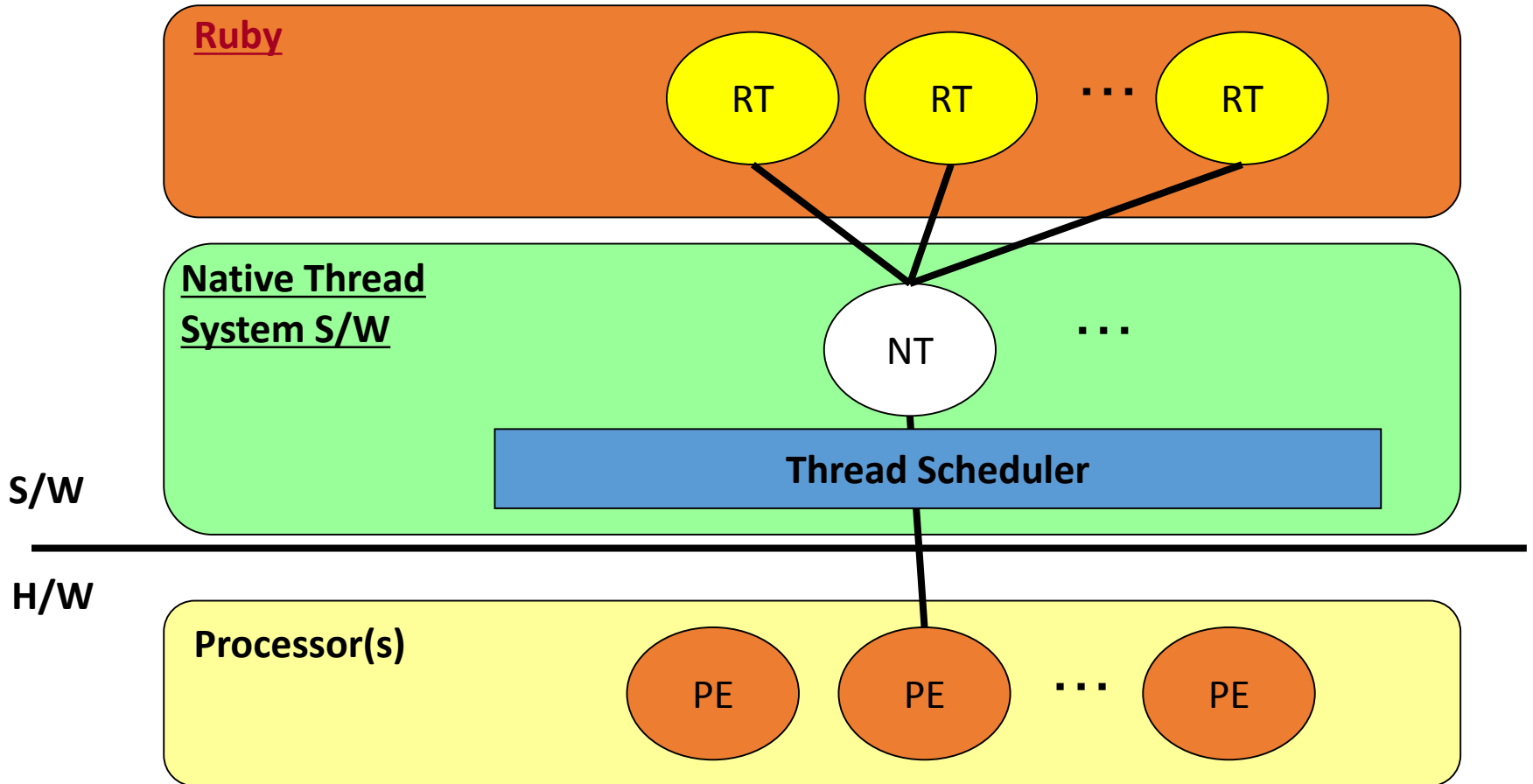
Ruby 1.8 and before

One OS (native) thread manages all Ruby threads
This technique is a.k.a. Green Thread



Threading

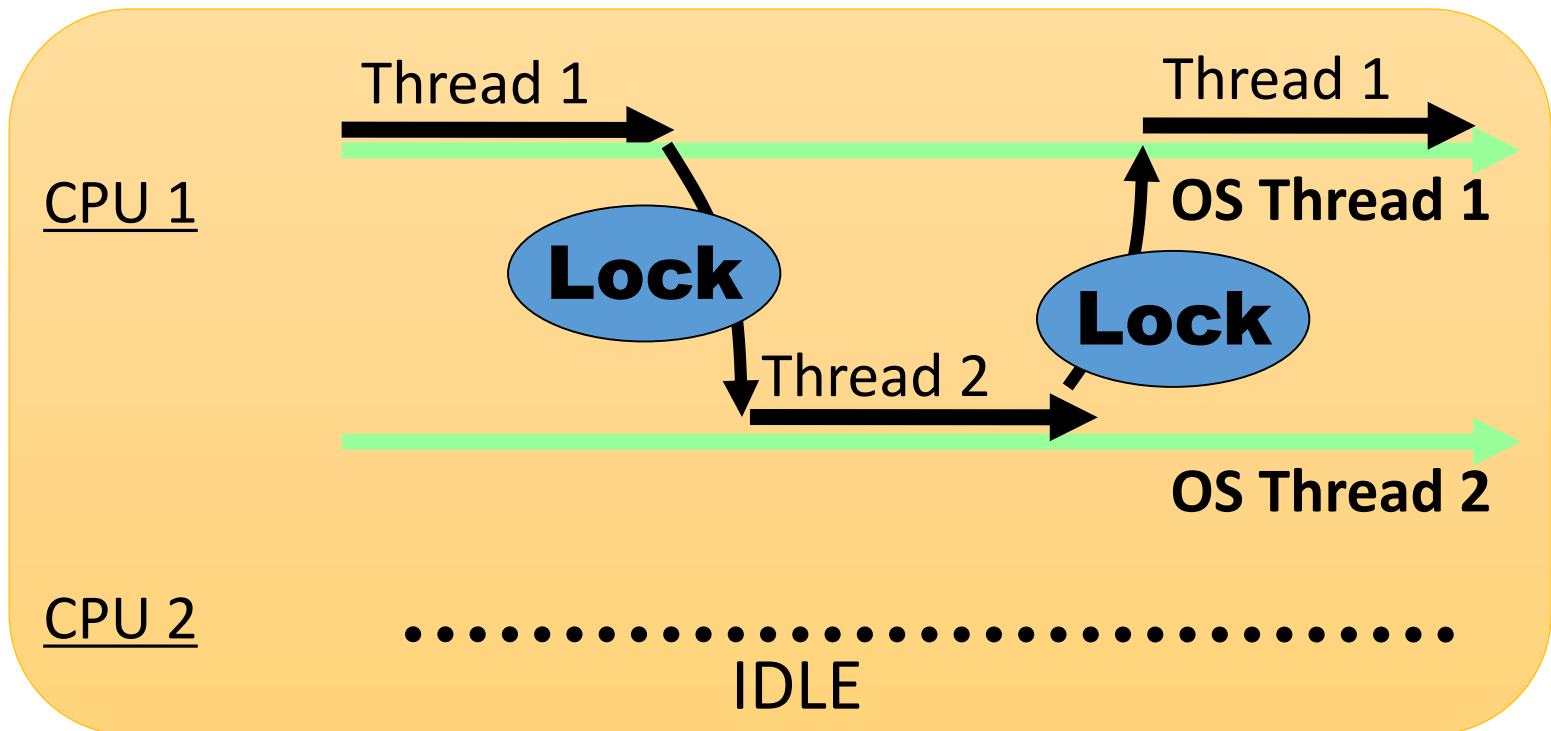
Layered view



Threading

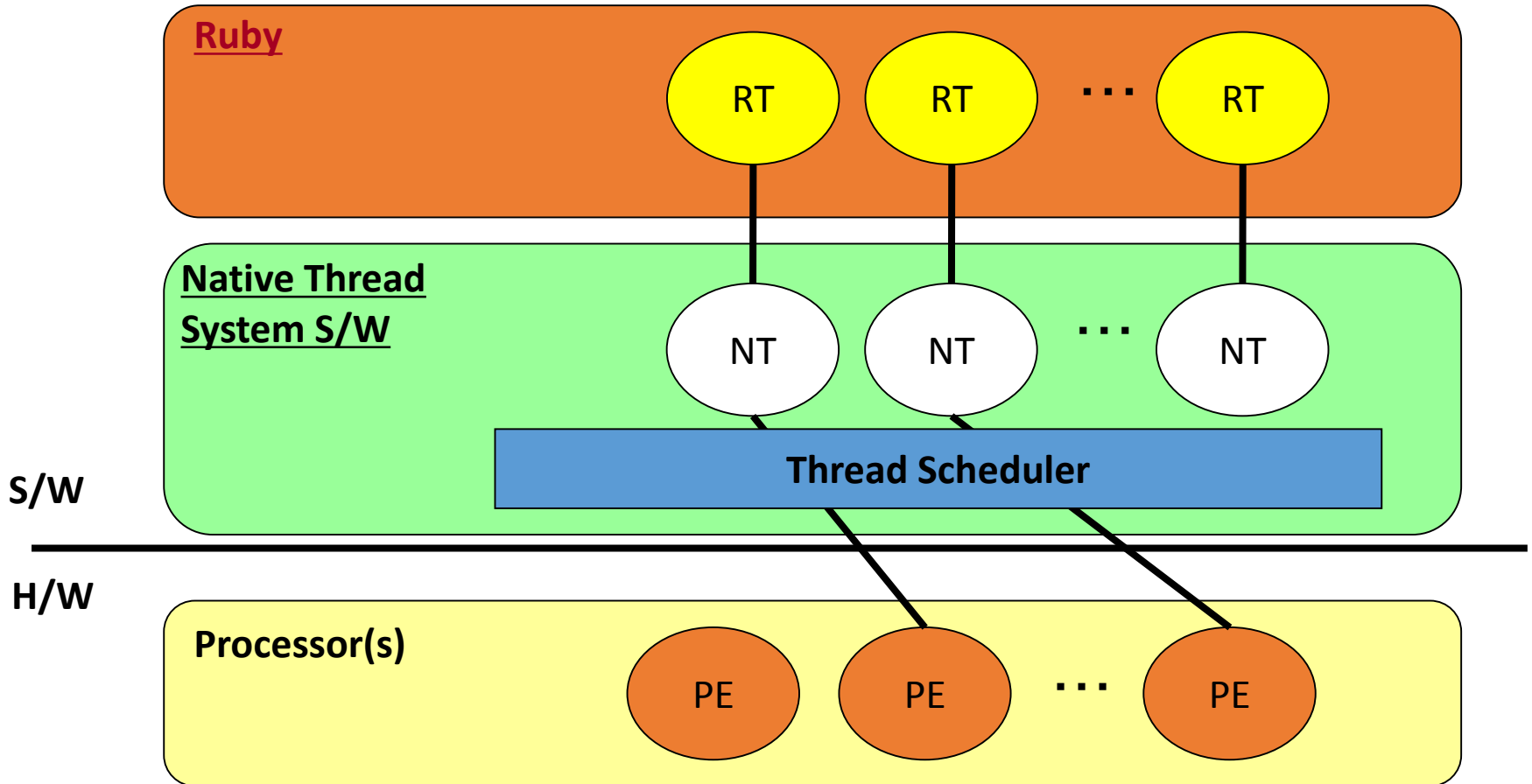
Ruby 1.9 and later

Native threads with Giant VM Lock



Threading

Layered view



Threading

Why not green threads?

- Advantage of green threads

- Lightweight creation

We don't need to make threads frequently.
(and we also have Fiber)

- Disadvantage of green threads

- Slow context switching (under portable way)
- Need to take care for blocking methods
 - Such as network read/write
- Difficult to collaborate with other C libraries using threads

Threading

Why GVL?

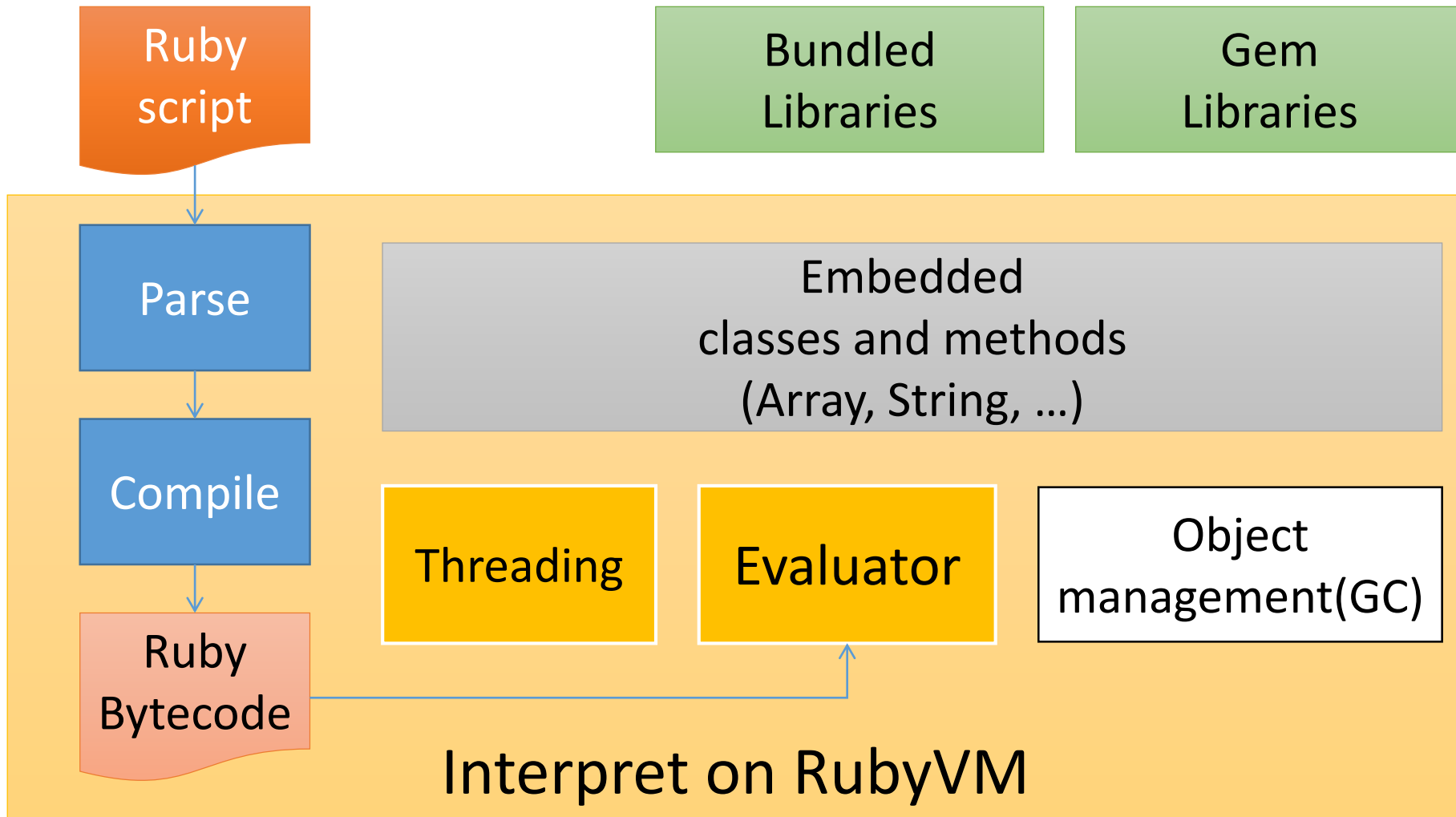
- To protect Ruby users from nightmare debugging
 - Shared parallel threading can make non-deterministic bugs which is too hard to debug
 - “Thread programming is too difficult for human being”
- Disadvantage
 - CRITICAL ISSUE: No parallel programming in Ruby
 - Need another programming model for parallel
 - Current ***SHARED EVERYTHING*** model is not match
 - Correct isolation level for each parallel execution units

Threading

How to make parallel ruby program?

- Now:
 - Use parallel threads provided by JRuby/Rubinius
 - If you think you can make correct thread programs
 - Use process (for example, w/ parallel gem, w/ dRuby)
- Future:
 - Introduce smart conventions to avoid threading bugs
 - Matz likes Actor model (Erlang)
 - Introduce limited shared memory model
 - Introduce smart debugging feature
 - Detecting bugs, avoid nondeterministic behaviors, ...

Object management (GC)



Object and memory management

- “Object.new” allocate a new object
 - “foo” (string literal) also allocate a new object
 - Everything are objects in Ruby!
- We don't need to “**de-allocate**” objects manually

Garbage collection

The automatic memory management



FIG. 109. — A GARBAGE COLLECTOR.

<http://www.flickr.com/photos/circasassy/6817999189/>

Growing the Ruby interpreter, Koichi Sasada,
RubyConf Brasil 2014

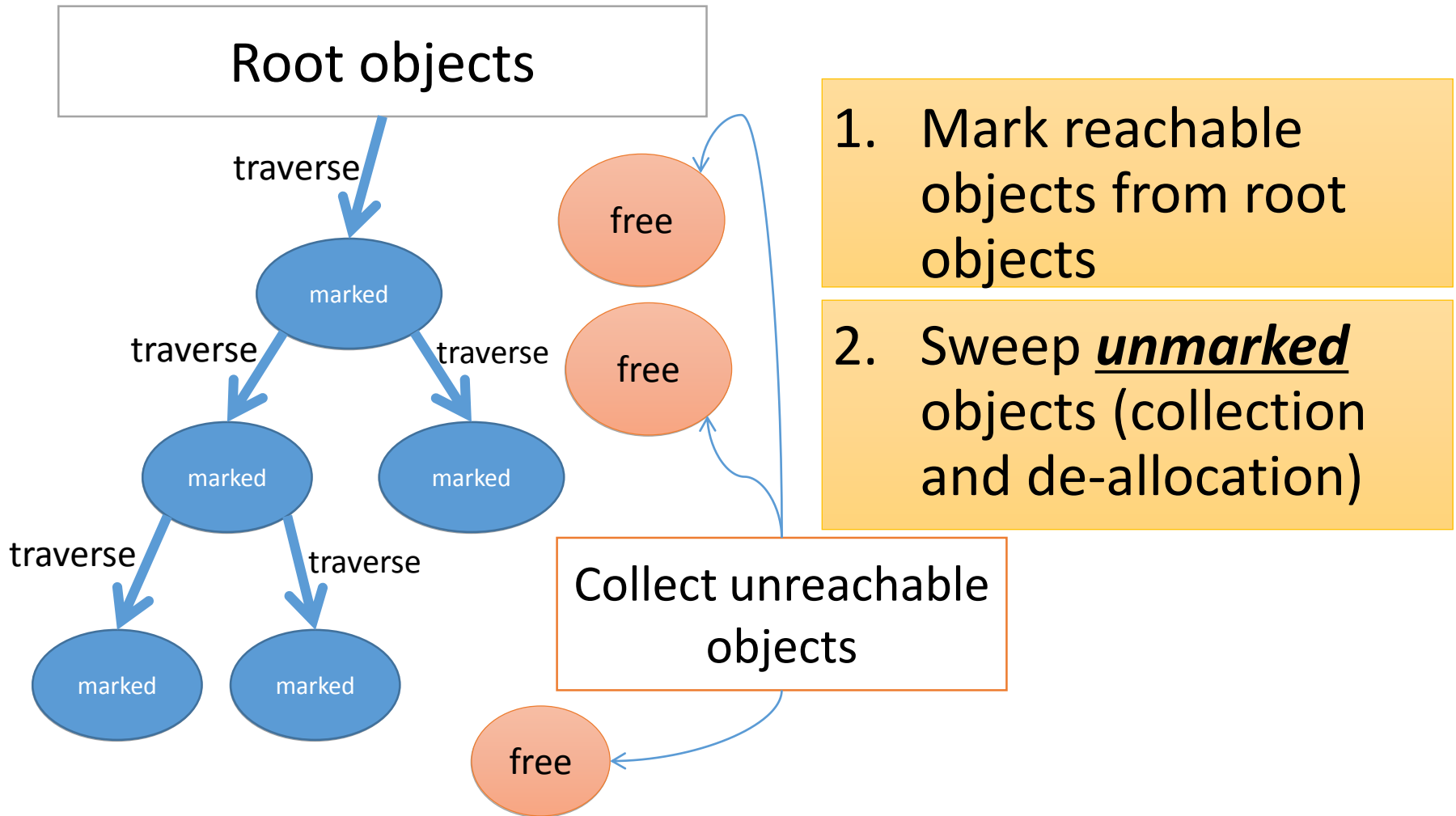
Automatic memory management

Basic concept

- **Garbage collector recycled “unused” objects automatically**



Mark & Sweep algorithm

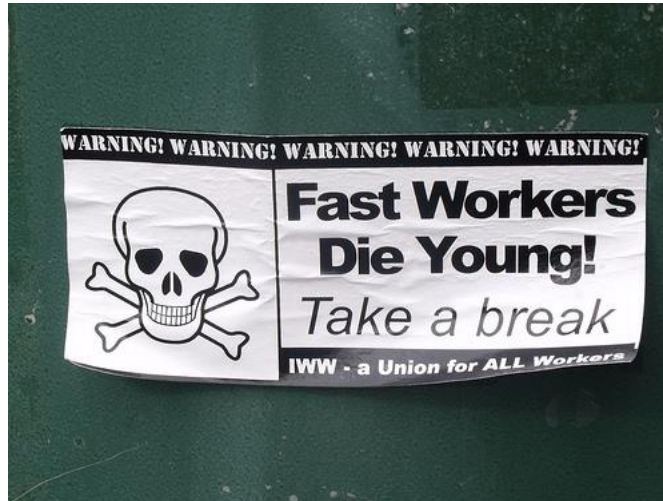


RGenGC: Restricted Generational GC

Generational GC (GenGC) from Ruby 2.1

- Weak generational hypothesis:

“Most objects die young”

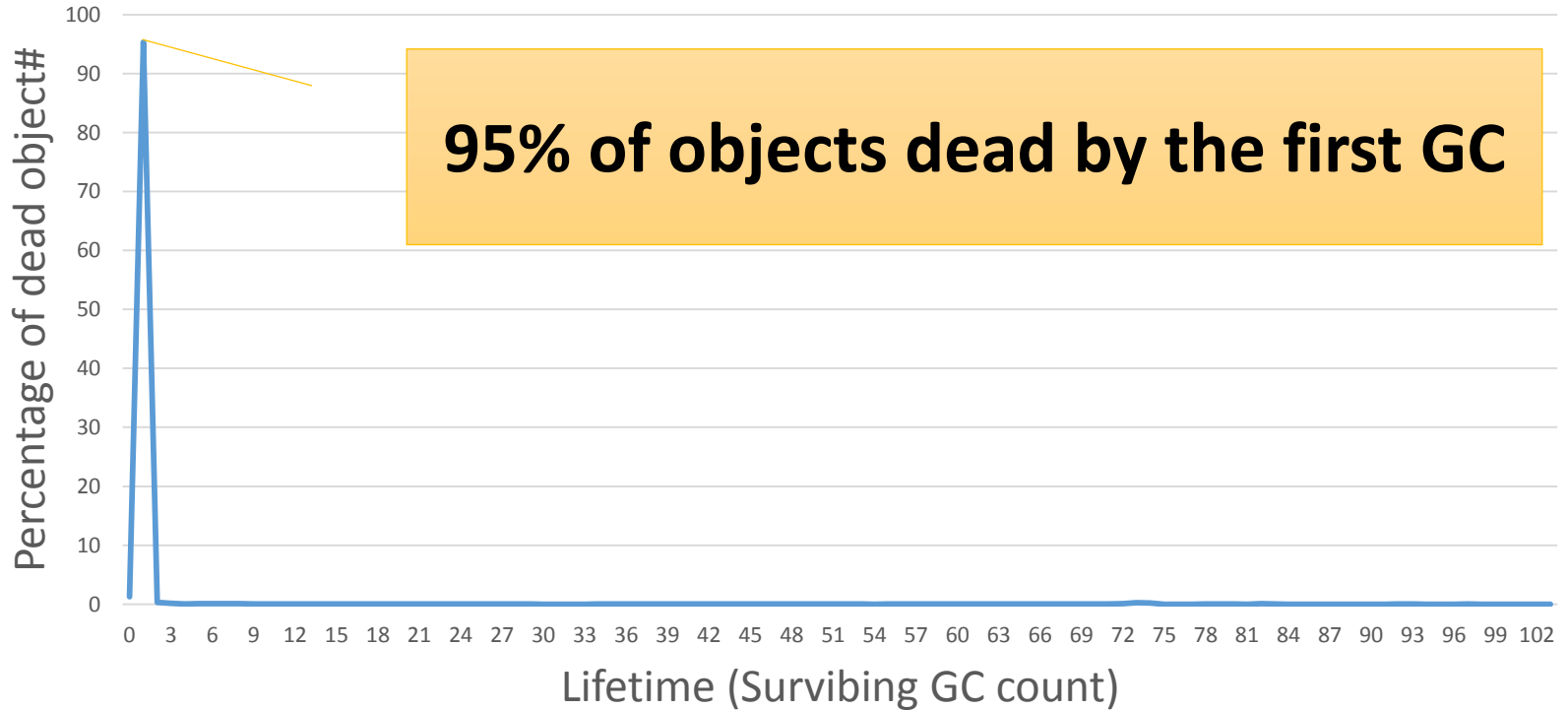


<http://www.flickr.com/photos/ell-r-brown/5026593710>

**→ Concentrate reclamation effort
only on the young objects**

Generational hypothesis

Object lifetime in RDoc
(How many GCs surviving?)



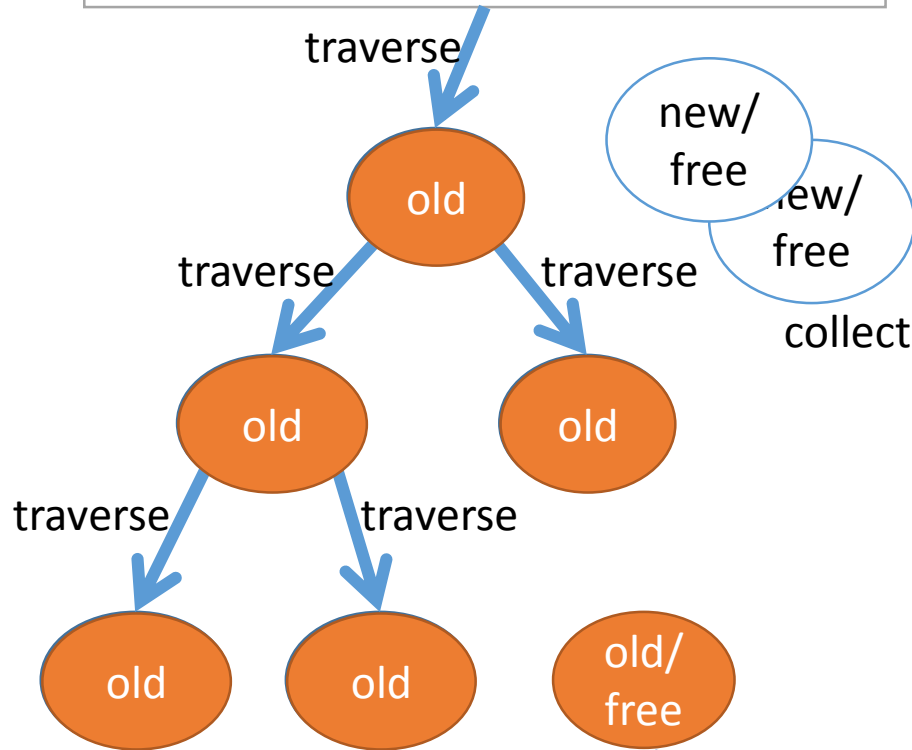
Generational GC (GenGC)

- Separate young generation and old generation
 - Create objects as young generation
 - Promote to old generation after surviving *n-th* GC
 - In CRuby, $n == 1$ (after 1 GC, objects become old)
 - $n == 2$ or 3 from Ruby 2.2
- Usually, GC on young space (minor GC)
- GC on both spaces if no memory (major/full GC)

GenGC [Minor M&S GC] (1/2)

1st MinorGC

Root objects



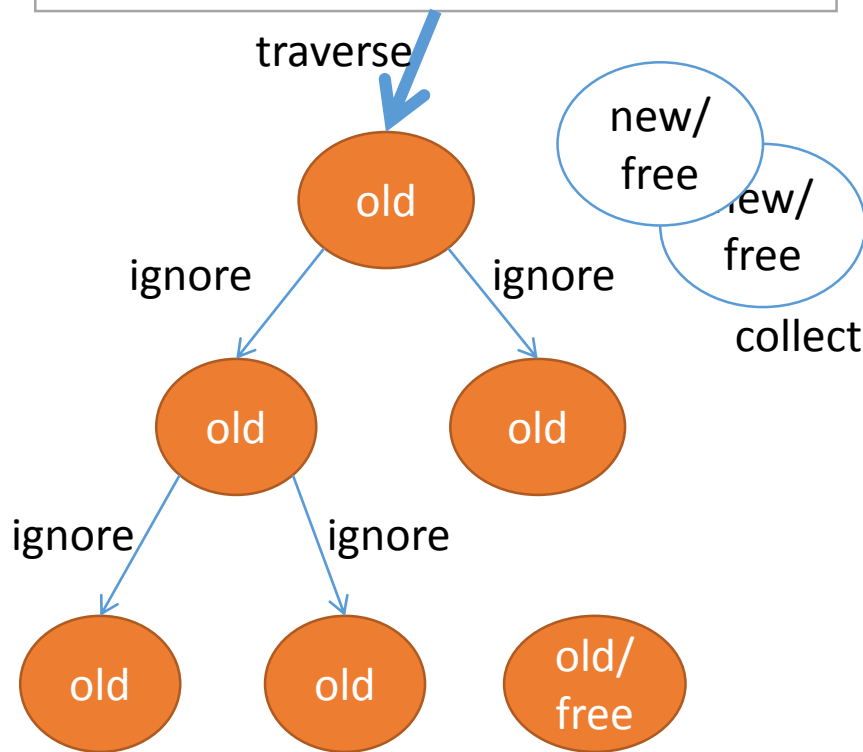
- Mark reachable objects from root objects.
 - Mark and **promote to old generation**
 - Stop traversing after old objects
- **→ Reduce mark overhead**
- Sweep not (marked or old) objects
- Can't collect Some unreachable objects

Don't collect old object even if it is unreachable

GenGC [Minor M&S GC] (2/2)

2nd MinorGC

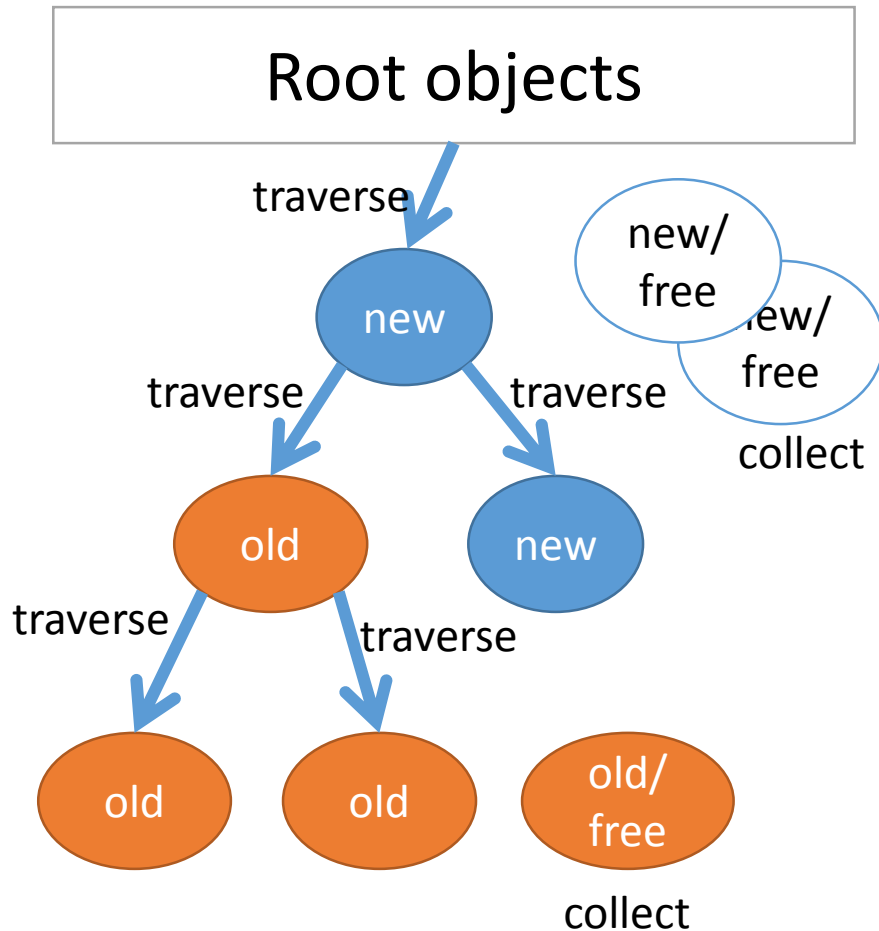
Root objects



- Mark reachable objects from root objects.
 - Mark and **promote to old generation**
 - Stop traversing after old objects
- **→ Reduce mark overhead**
- Sweep not (marked or old) objects
- Can't collect Some unreachable objects

Don't collect old object even if it is unreachable

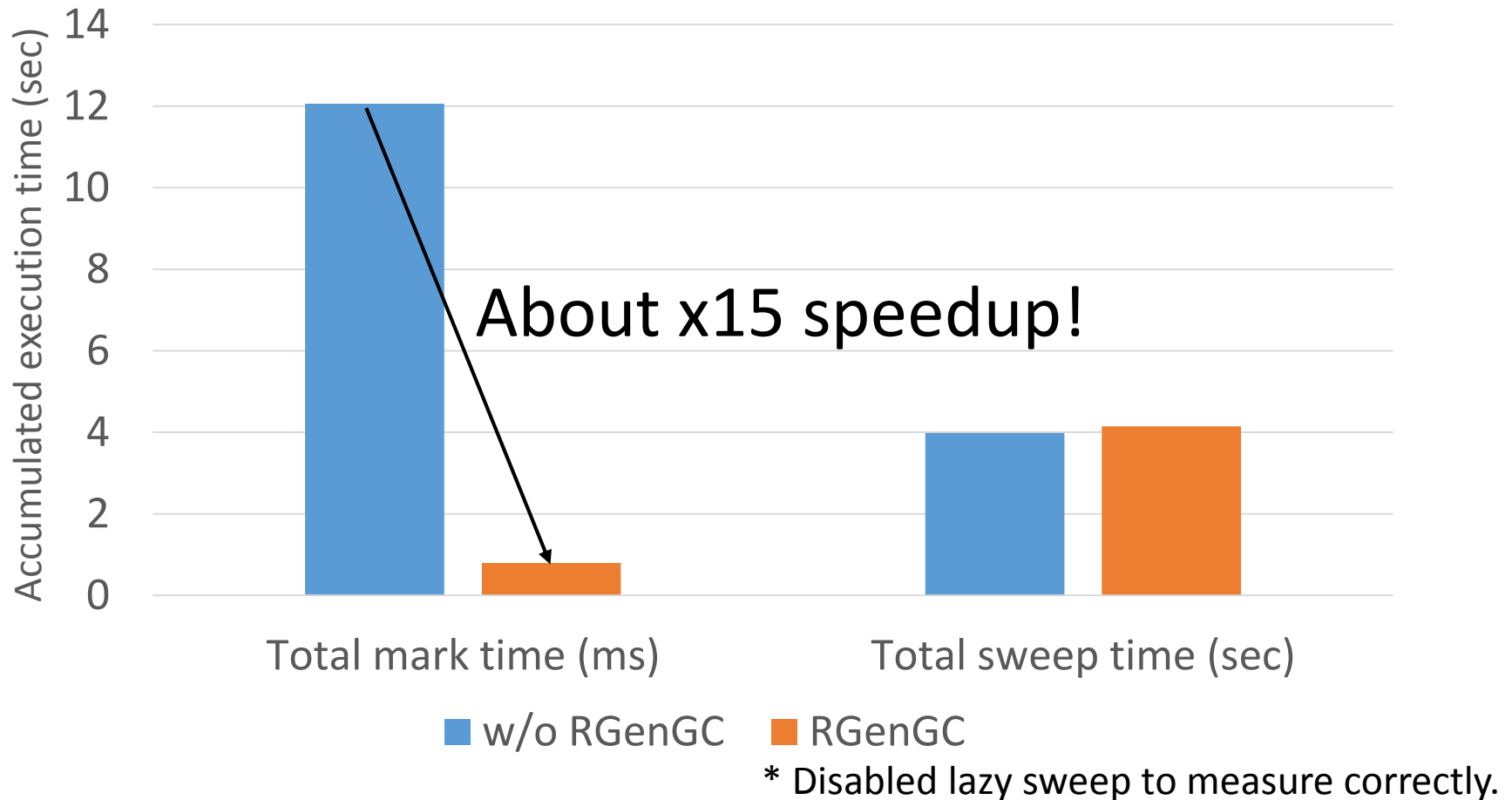
GenGC [Major M&S GC]



- Normal M&S
- Mark reachable objects from root objects
 - Mark and **promote to old gen**
- Sweep unmarked objects
- Sweep all unreachable (unused) objects

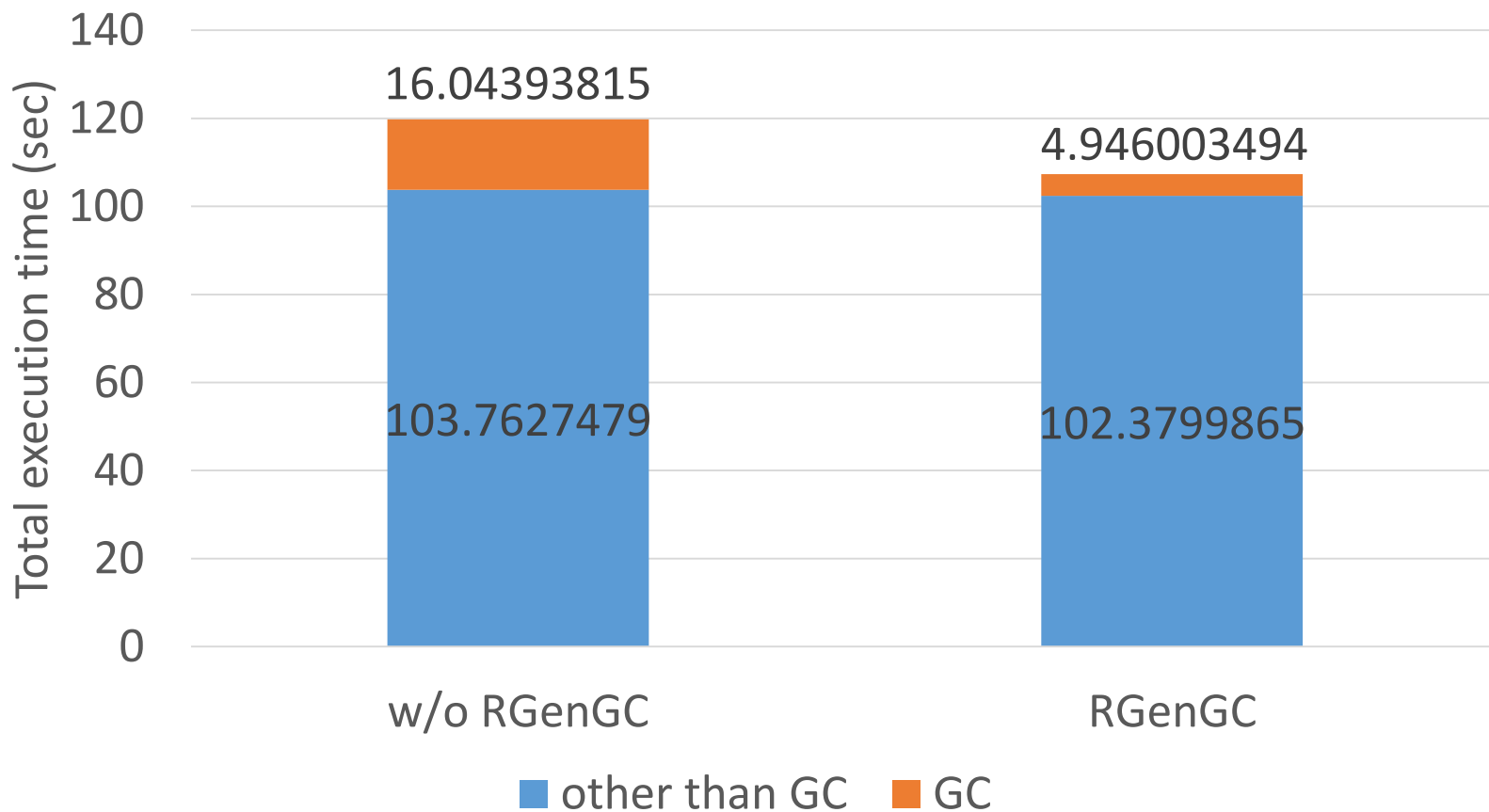
RGenGC

Performance evaluation (RDoc)



RGenGC

Performance evaluation (RDoc)



* 12% improvements compare with w/ and w/o RGenGC

* Disabled lazy sweep to measure correctly.

RincGC:

Restricted incremental GC

RincGC

Background and motivation

- Ruby 2.1 had introduced generational GC
 - Short marking time on minor GC
 - Improve application throughput
- Still long pause time on major GC
 - Long pause time affects user response time

Proposal:

RincGC: Incremental GC for major GC

- Introducing incremental GC to reduce pause time
- Can combine with Generational GC

| | Generational GC | Incremental GC | Gen+Inc GC |
|------------|-----------------|------------------|------------|
| Throughput | High | Low (a bit slow) | High |
| Pause time | Long | Short | Short |

RincGC: Base idea

Incremental GC algorithm

- Move forward GC processes incrementally
 - Mark slots incrementally
 - Sweep slots incrementally
- Incremental marking in 3 phase
 - (1) Mark roots (pause)
 - (2) Mark objects reachable from roots (incremental)
 - (3) Mark roots again, and mark remembered objects (pause)
- Mark objects with three state (white/grey/black)
 - White: Untouched objects
 - Grey: Marked, and prepare to mark directly reachable objects
 - Black: Marked, and all directly reachable objects are marked
- Use write barriers to avoid marking miss from marked objects to live objects
 - Detect new reference from black objects to white objects
 - Remember such source black objects (marked at above (3))

RincGC:

Incremental GC for CRuby/MRI

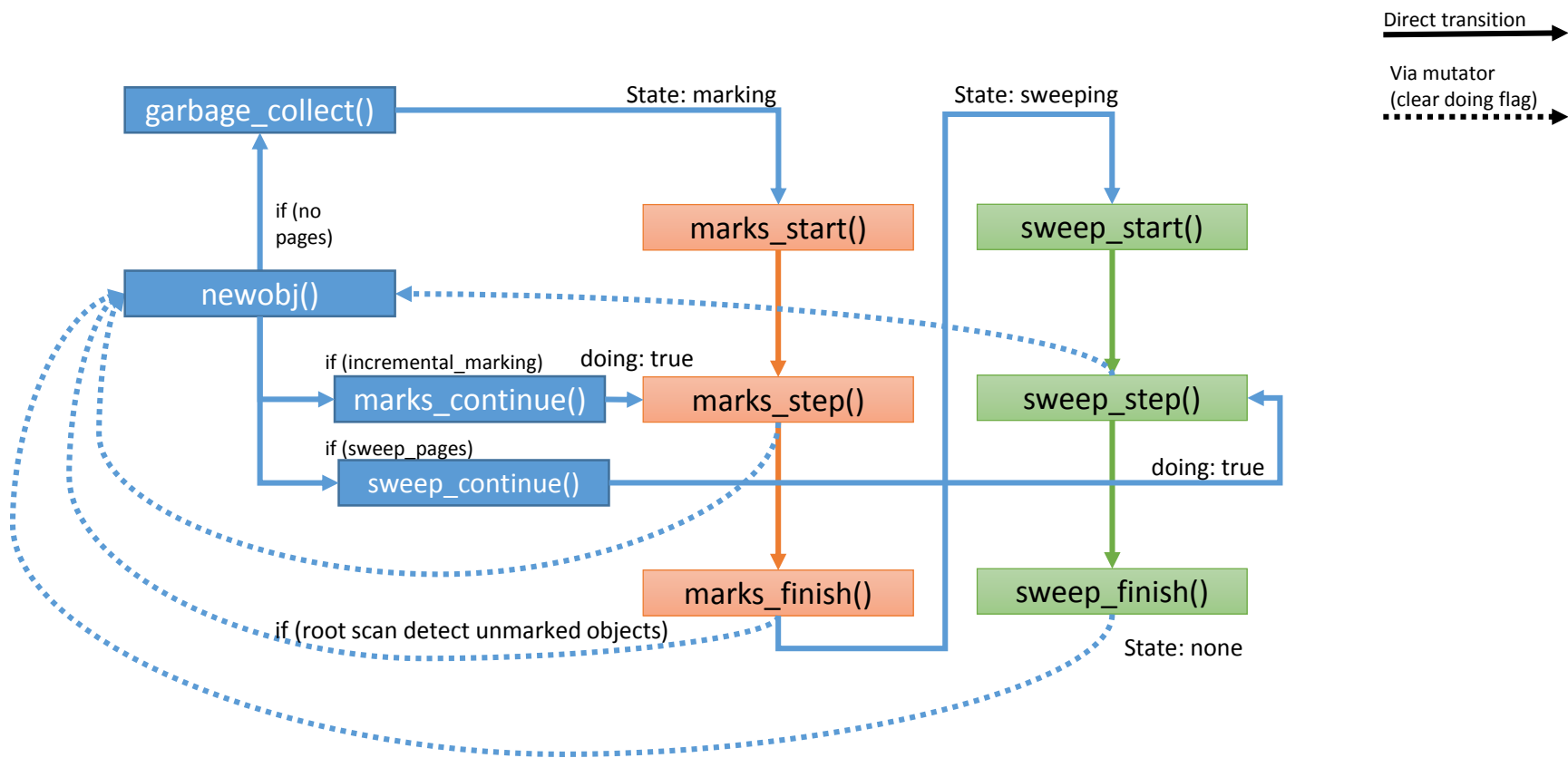
- Incremental marking
 - (1) mark roots (`gc_mark_roots()`)
 - (2) Do incremental mark at `rb_newobj_of()`
 - (3) Make sure write barrier with WB-protected objects
 - (4) Take care of **WB-unprotected objects** (MRI specific)
- Incremental sweeping
 - Modify current lazy sweep implementation

RincGC:

Incremental marking

- (1) mark roots (`gc_mark_roots()`)
 - Push all root objects onto “mark_stack”
- (2) Do incremental mark at `rb_newobj_of()`
 - Fall back incremental marking process periodically
 - Consume (pop) some objects from “mark_stack” and make forward incremental marking
- (3) Make sure write barrier with WB-protected objects
 - Mark and push pointed object onto “mark_stack”
- (4) Take care of **WB-unprotected objects** (MRI specific)
 - After incremental marking (“mark_stack” is empty), re-scan all roots and all living non-WB-protected objects
 - WB-unprotected objects are represented by bitmap (`WB_UNPROTECTED_BITS`)

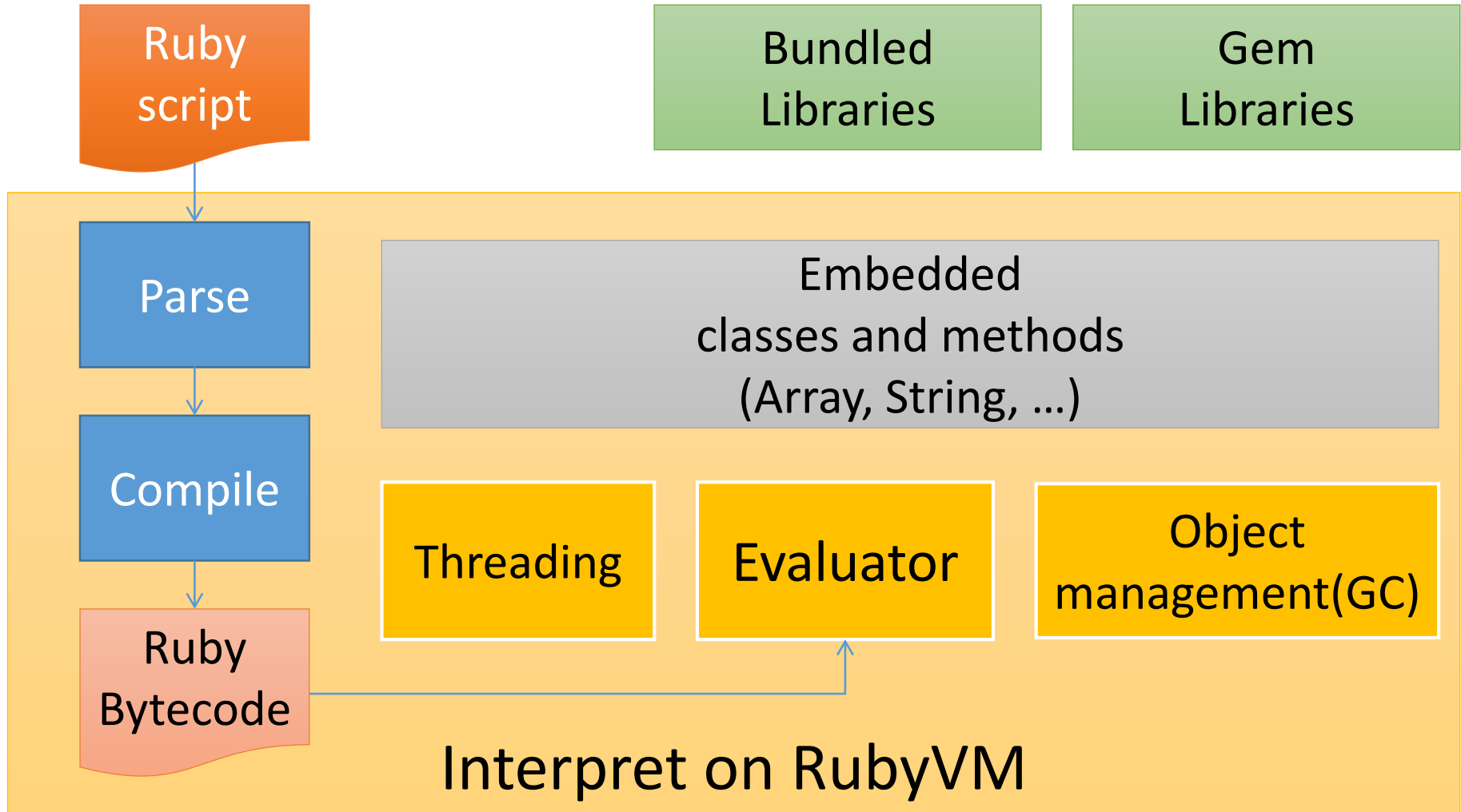
RincGC: Diagram



Growing up the Ruby Interpreter

How do we grow up the Ruby interpreter?

DO EVERYTHING! NO SILVER BULLET!



DO EVERYTHING! NO SILVER BULLET!

Loop do

- Survey techniques
- Implement techniques
- Invent new techniques
- Evaluate techniques

end # endless

**DO EVERYTHING!
NO SILVER BULLET!**

We did.

We are doing.

We will do!!

**Only continuous effort
improves software quality.**

Future work: Many many many!!

- Evaluator
 - JIT compilation
 - More drastic optimizations
- Threading
 - Parallel execution model (not a threading?)
- Object management and GC
 - Compaction GC
 - Lightweight object allocation
 - CoW friendly memory management
- And more

Summary

- Ruby 2.1 and Ruby 2.2
- How to grow up the Ruby interpreter?
 - Evaluator
 - Threading
 - Object management / Garbage collection

Summary

- Ruby 2.1 and Ruby 2.2
- How to grow up the Ruby interpreter?

My answers is:

#=> Continue software development
(with love?)

Thank you for your attention

Koichi Sasada

<ko1@heroku.com>

