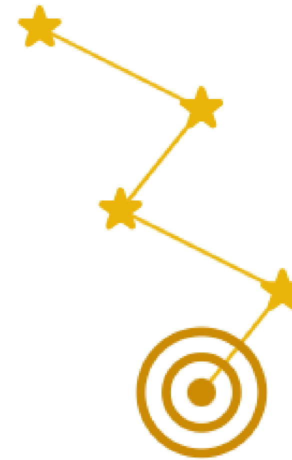
The AST Galaxy (responds) to the Virtual Machine Blues

Koichi Sasada
STORES, Inc

About this talk

- A new technique for building fast interpreters –
  ASTro: AST-based Reusable Optimization Framework
  - Generates **fast native code** from a tree-walking interpreter
  - Not limited to Ruby, but Ruby is the primary target
  - Still in the research phase
- Motivations:
  the growing Complexity of Ruby
- How ASTro works
- Early evaluation results



ASTro

AST-based

Reusable Optimization

Framework

abruby: a bit Ruby on ASTro

abruby

a bit Ruby

Koichi Sasada

- Ruby interpreter developer at **STORES, Inc.** (since 2023), working with @mametter
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - M:N Thread scheduler (Ruby 3.3~)
  - …
- Director of the Ruby Association (since 2012)
- Owner of @. Bookstore (since 2024)

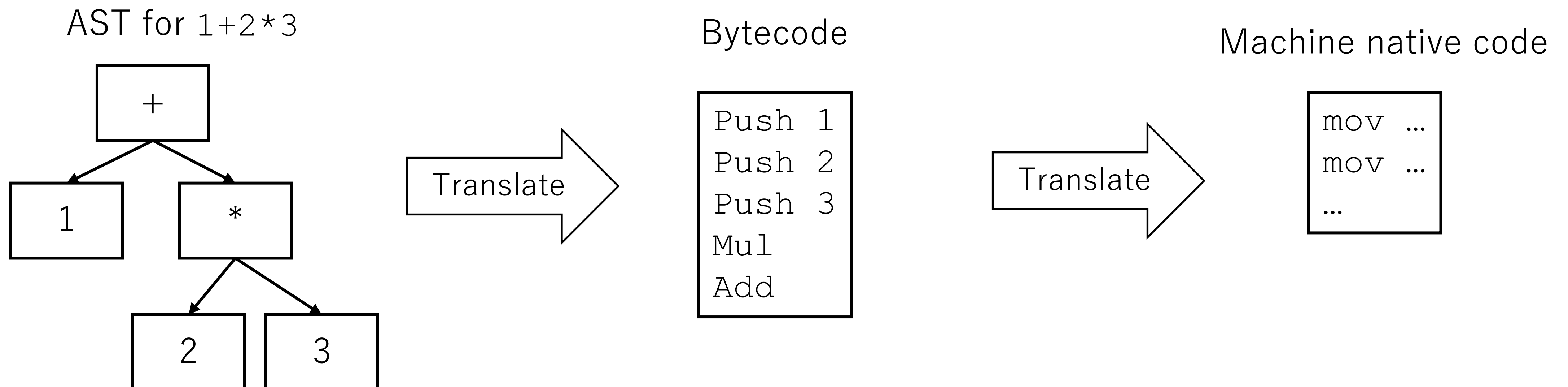
Take the quiz & get swag!
Talk to the STORES staff.

History

YARV: Yet Another Ruby Virtual machine

- Ruby 1.8: Tree-walking interpreter
  - Traverses AST nodes and executes them
- Ruby 1.9: Bytecode interpreter
  - Translates the AST into stack-machine-based bytecode
- Ruby 3.1: JIT compilation

We made Ruby faster, but every step also made the implementation heavier.



Motivation (1) Implementation complexity

Source code —Parsing→ AST —Compiling→ Bytecode —JIT compiling→ Native code

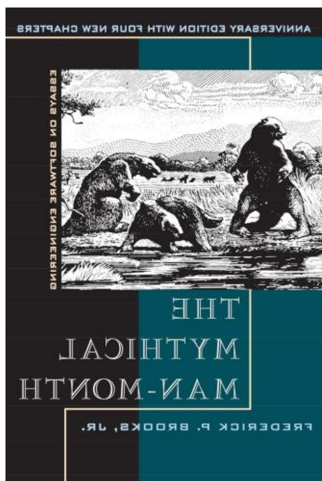
Tree-walking Interpreter —Executes→ AST

~Ruby 1.8

Virtual machine —Executes→ Bytecode

Ruby 1.9~ (YARV)

Ruby 3.1~ (YJIT/ZJIT)



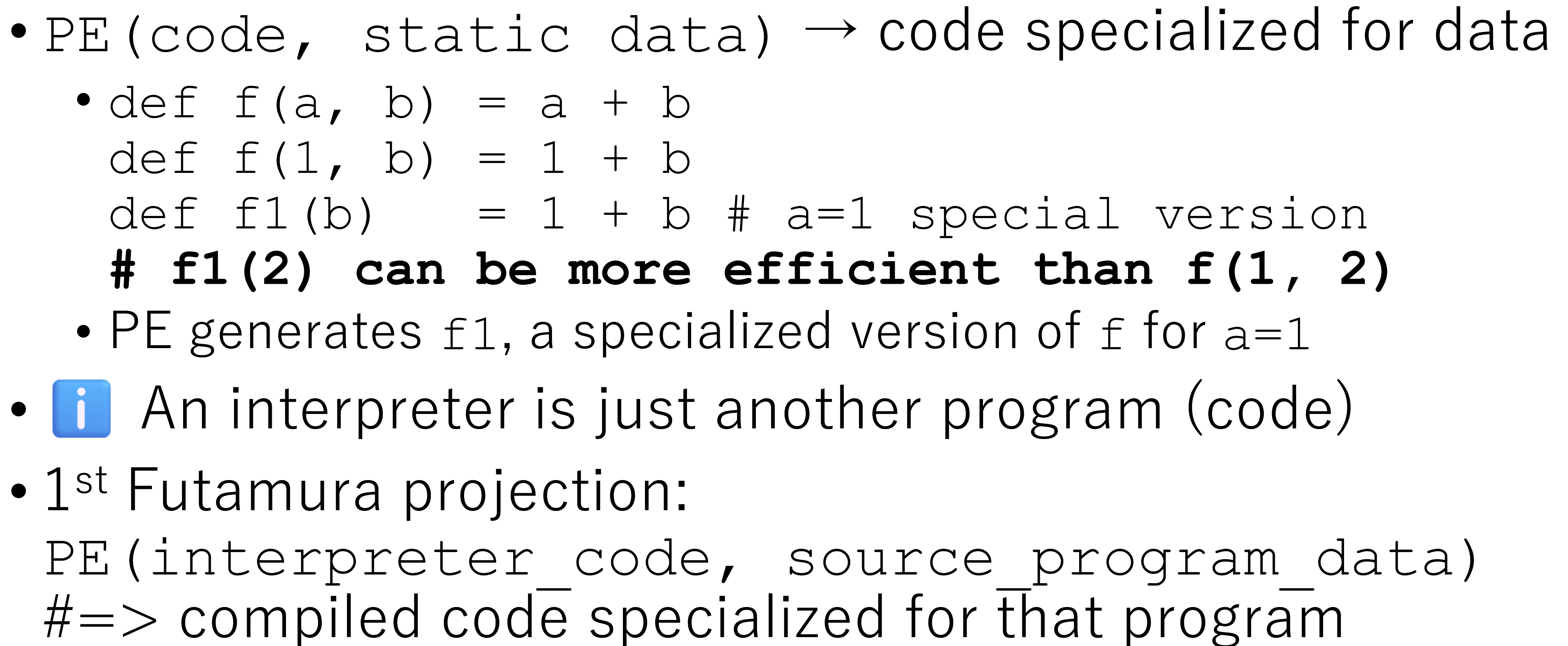
✅ Simple, Easy
❌ Slow

⟷

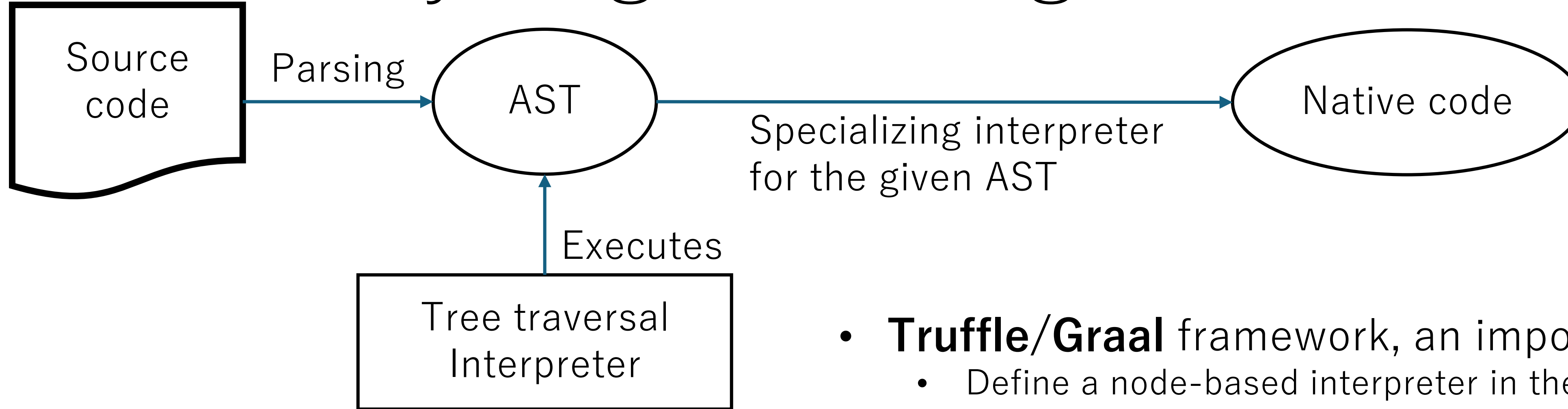
✅ Fast
❌ Complicated

This is a kind of virtual machine blues…

Partial Evaluation (PE)



Motivation (2) Partial Evaluation, but Heavyweight Existing Toolchains



✅ Simple, Easy and Fast!!
❌ Heavyweight Framework
Hard to make and modify

- **Truffle/Graal** framework, an important system:
  - Define a node-based interpreter in the Truffle DSL
  - Profile execution of those nodes
  - Specialize (PE) the given AST and interpreter using Truffle framework and generate native code with Graal compiler
  - **Built on a large Java-based framework**

❓ Can we make a lightweight PE framework?

We need a tool that can
generate highly optimized native code
for **widely available**
and **easy to use.**
Do you know?

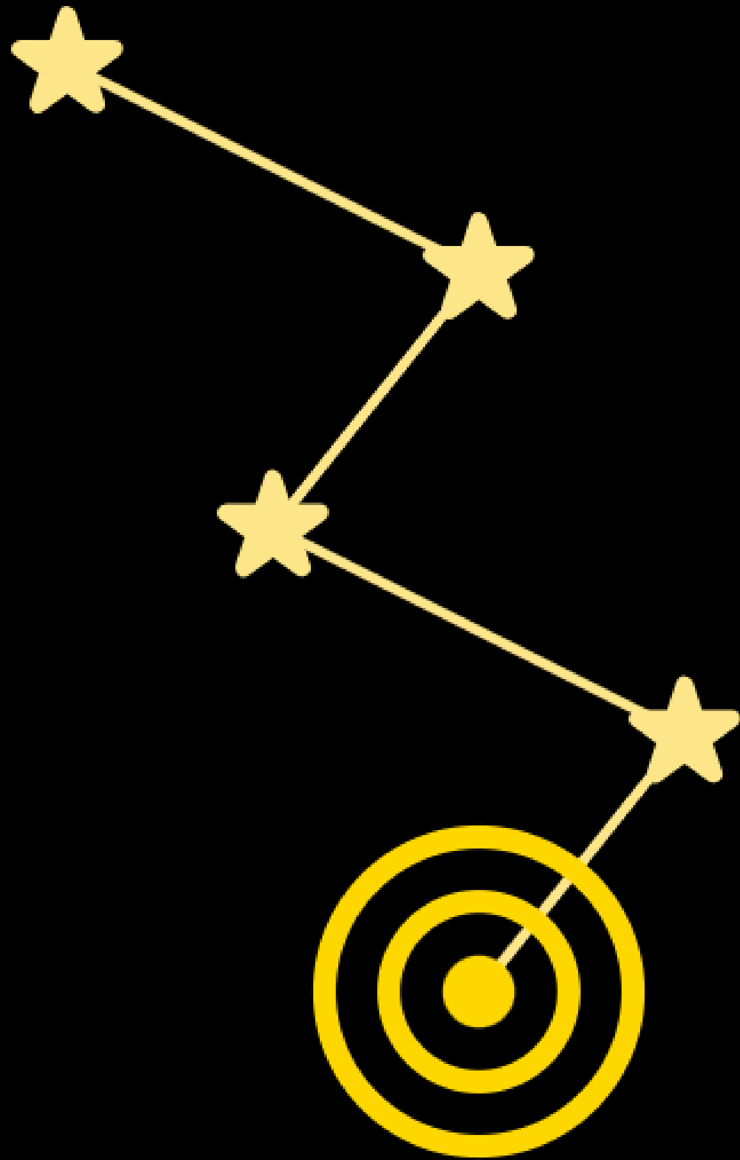
❓ Can we make a lightweight PE framework?

We need a tool that can
generate highly optimized native code
for **widely available**
and **easy to use.**
Do you know?

C compilers!

Most systems already have a C compiler

https://github.com/ko1/astro/



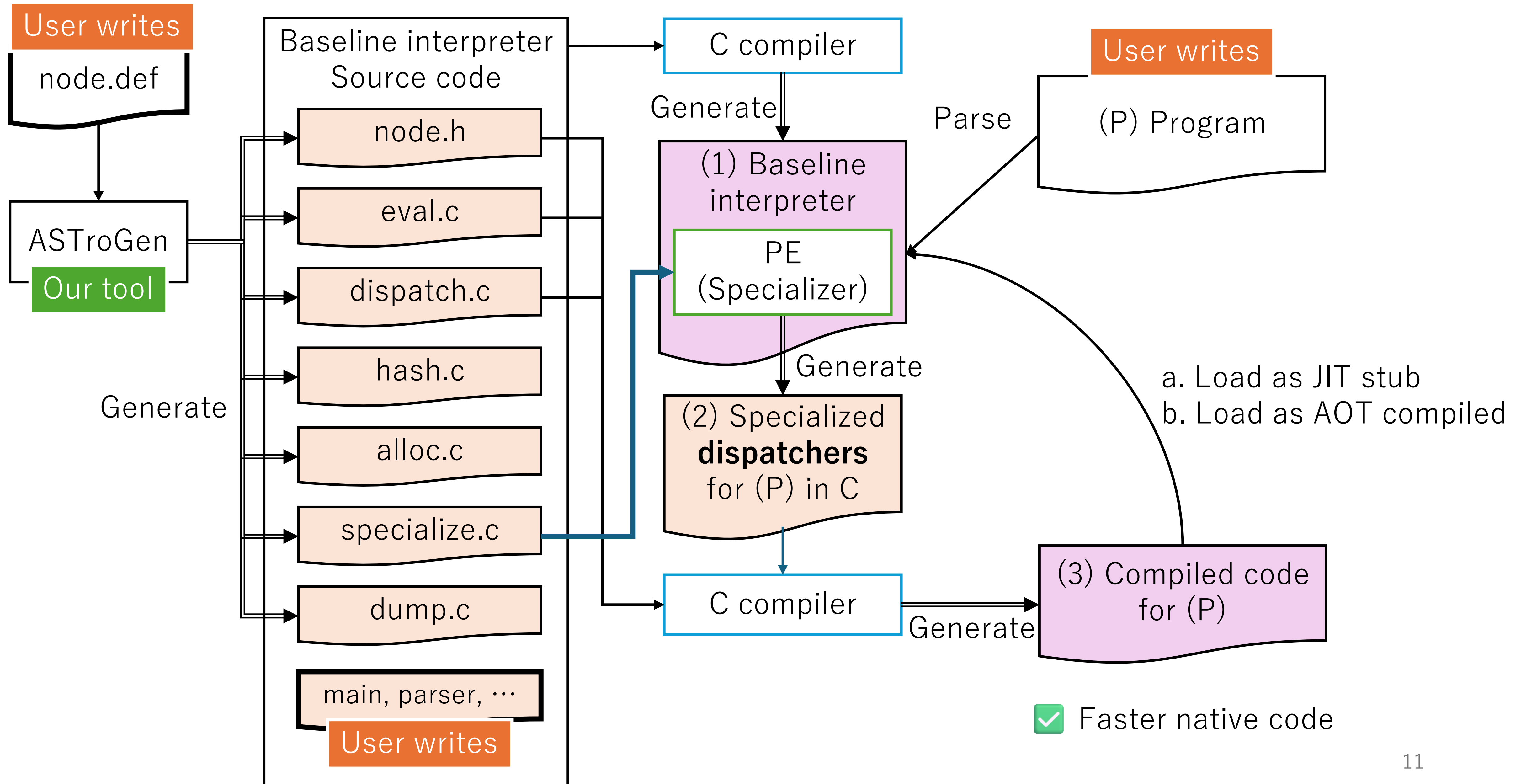
ASTro

AST-based

Reusable Optimization

Framework

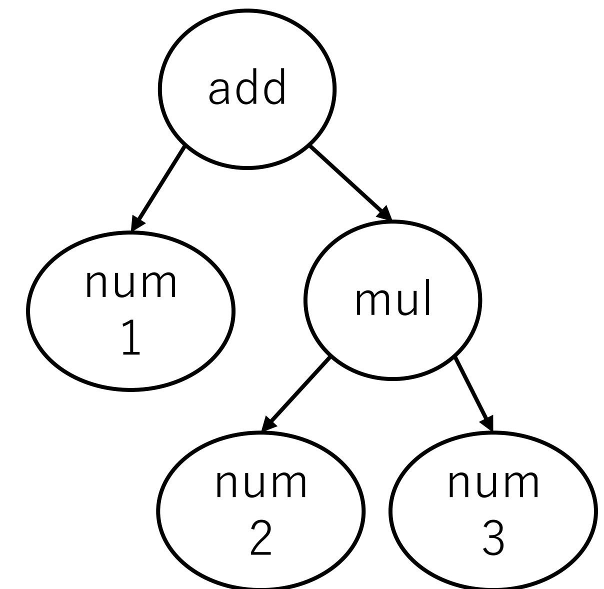
ASTro workflow



Example: Tiny Calc language (3 node types)

A minimal interpreter definition in node.def

```
// node.def
NODE_DEF             // start of the definition
node_num(int n)  // node name & attributes
{ return n; }      // node behavior
NODE_DEF
node_add(NODE *lv, NODE *rv)
{ return EVAL_ARG(lv) + EVAL_ARG(rv); }
NODE_DEF
node_mul(NODE *lv, NODE *rv)
{ return EVAL_ARG(lv) * EVAL_ARG(rv); }
```



Separating **evaluators** and **dispatchers** Leaf node case

```
NODE_DEF
node_num(int n)
{ return n; }
```

Language-specific context

```
static VALUE
EVAL_node_num(CTX *c, NODE *n, int num)
{ return num; }
```

Copies the behavior definition directly

```
static VALUE
DISPATCH_node_num(CTX *c, NODE *n)
{
  return EVAL_node_num(c, n, n->u.node_num.num);
}
```

generate

Pass the extracted value to the eval function

Separating evaluators and dispatchers
Internal node case

```
NODE_DEF
node_add(NODE *lv, NODE *rv)
{ return EVAL_ARG(lv) + EVAL_ARG(rv); }
```

```
VALUE EVAL_node_add(CTX *c, NODE *n,
  NODE *lv, node_dispatcher_func_t lv_disp,
  NODE *rv, node_dispatcher_func_t rv_disp)
{
  return EVAL_ARG(c, lv) + EVAL_ARG(c, rv);
}
```

Receives **child node** together with their **dispatchers**, which are used in the `EVAL_ARG()` (simply calls dispatchers)

```
#define EVAL_ARG(c, n)   (*n##_disp)(c, n)
```

```
VALUE DISPATCH_node_add(CTX *c, NODE *n)
{
  return EVAL_node_add(c, n,
                       n->u.node_add.lv, disp(n->u.node_add.lv),
                       n->u.node_add.rv, disp(n->u.node_add.rv));
}
```

Passes the extracted child nodes and corresponding dispatchers to the evaluator

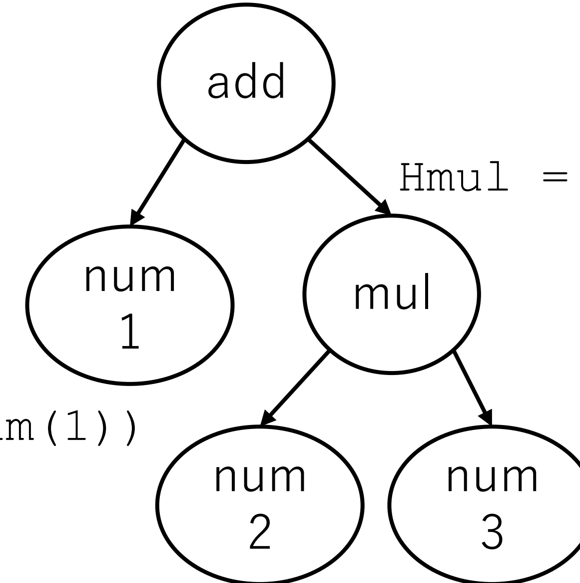
ASTro's important Idea 1:

Naming AST subtrees with Merkle Hashes

- Merkle hash
  - Compute each node's hash from its own value and the hashes of its children
- We use the hash value as a stable name for an AST subtree **across runs.**

`1 + 2 * 3`

`Hadd = hmerge("add", H1, Hmul)`



`Hmul = hmerge("mul", H2, H3)`

`H1 = h(num(1))`

`H2 = h(num(2))`

`H3 = h(num(3))`

[Open Question] Which hash algorithm is sufficient (collision resistance vs speed)?

H… is something like "`fb75fdabb0d5ef6`" in practice

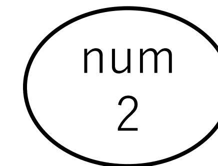
ASTro's important Idea 2:
Partial evaluation by generating dispatch functions

- ASTroGen generates a **partial evaluator** (PE)
  - PE emits **ONLY specialized dispatchers**
  - This is easy because we do not need to analyze or rewrite evaluator bodies
  - The C compiler can inline specialized dispatchers with evaluators

```
// specialized dispatcher for num(2)
SD_<H2>(CTX *c, NODE *n) {
  return EVAL_node_num(c, n, 2);
}
```

<H2> is the name of the node (hash value)

Generate specialized code for



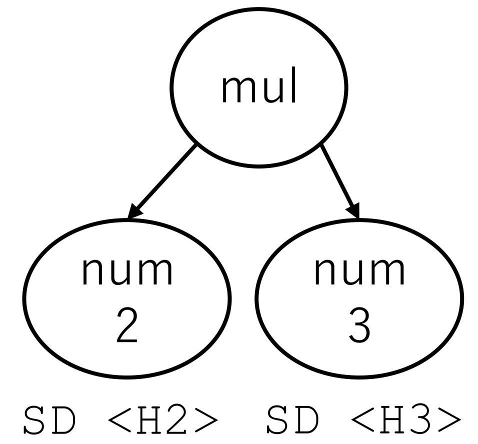
H2 = h(num(2))

Idea 2: Partial evaluation by generating dispatch functions

```
// specialized dispatcher for mul(num(2), num(3))
SD_<Hmul>(CTX *c, NODE *n)
{
  return EVAL_node_mul(c, n,
                       n->u.node_mul.lv, SD_<H2>,
                       n->u.node_mul.rv, SD_<H3>);
}
```

Specialized dispatch function pointers

Generate specialized code for



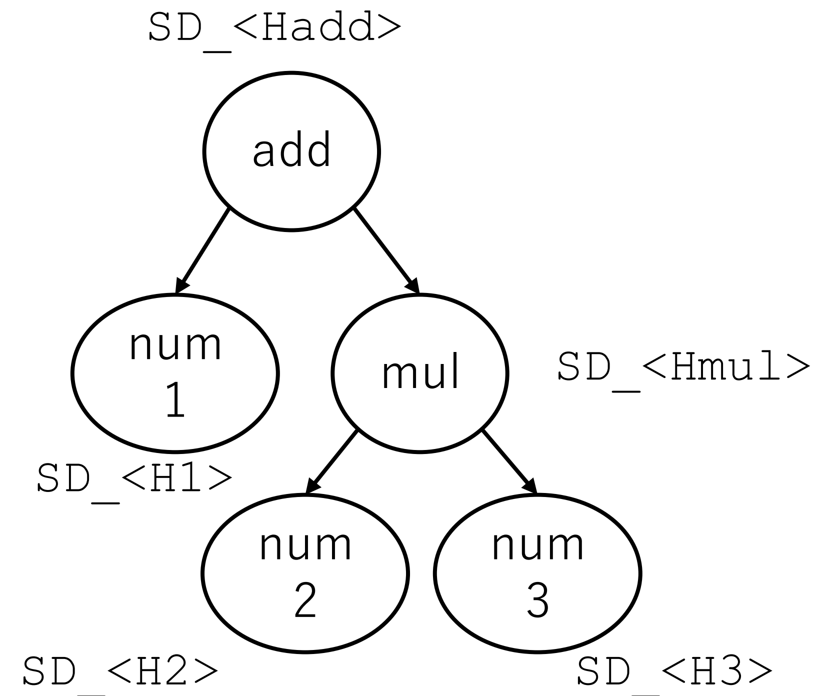
We generate specialized dispatchers recursively, and use them in their parent dispatchers

Because the dispatcher names are known statically, modern C compilers can inline them

Idea 2:
Partial evaluation by generating dispatch functions

```
0000000000001a20 <SD_<Hadd>>:
1a20: endbr64
1a24: mov $0x7,%eax ; literal 7
1a29: ret
```

The technique can generate extremely simple native code with C compiler inlining



See our paper for more details
ASTro: An AST-Based Reusable Optimization Framework
https://dl.acm.org/doi/10.1145/3759548.3763371

https://github.com/ko1/astro/tree/master/sample/abruby

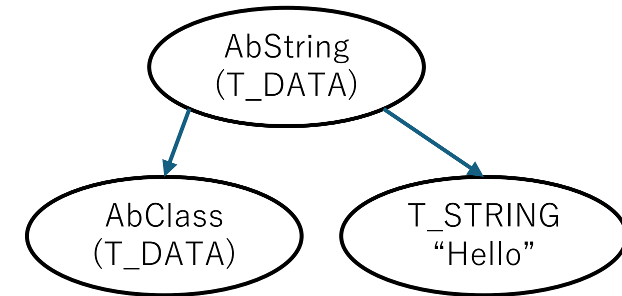
abruby: “a bit Ruby” on ASTro

- Supports many features
  - Object oriented (OO) features (Classes/Modules/GC)
  - Proc, Lambda
  - Variables, Constants
  - Exception handling features
  - Many built-in classes, at least to run optcarrot
- Commands
  - abruby – like ruby command
  - iabrb – like irb command

DEMO if I have a time

abruby implementation techniques

A string representation



- Implemented as a CRuby extension
  - Reuses many built-in features such as GC, String, …
  - Replacing YARV part with ASTro-based interpreter
  - abruby has its own OO mechanism
    - AbClass (method table, etc) for each object
    - All objects wrapped by T_DATA, so 2 objects are needed for a String
  - Commands are written naturally in Ruby
- Prism AST → abruby AST (defined in node.def)
- Exception handling by returning [value, state] values
  - Like Go language
  - No setjmp/longjmp
- Type-specific nodes for performance
  - Like opt_plus VM instruction

abruby development statistics

- About 4,000 lines of node.def
- About 6,000 lines of others
  - Most of the implementation is a wrapper around CRuby code
- Contributors – Thank you!!
  - Claude code for a writer
  - Codex for a review
  - … and Me (just asking)
- 1 week to make almost interpreter
- 1 week to optimize

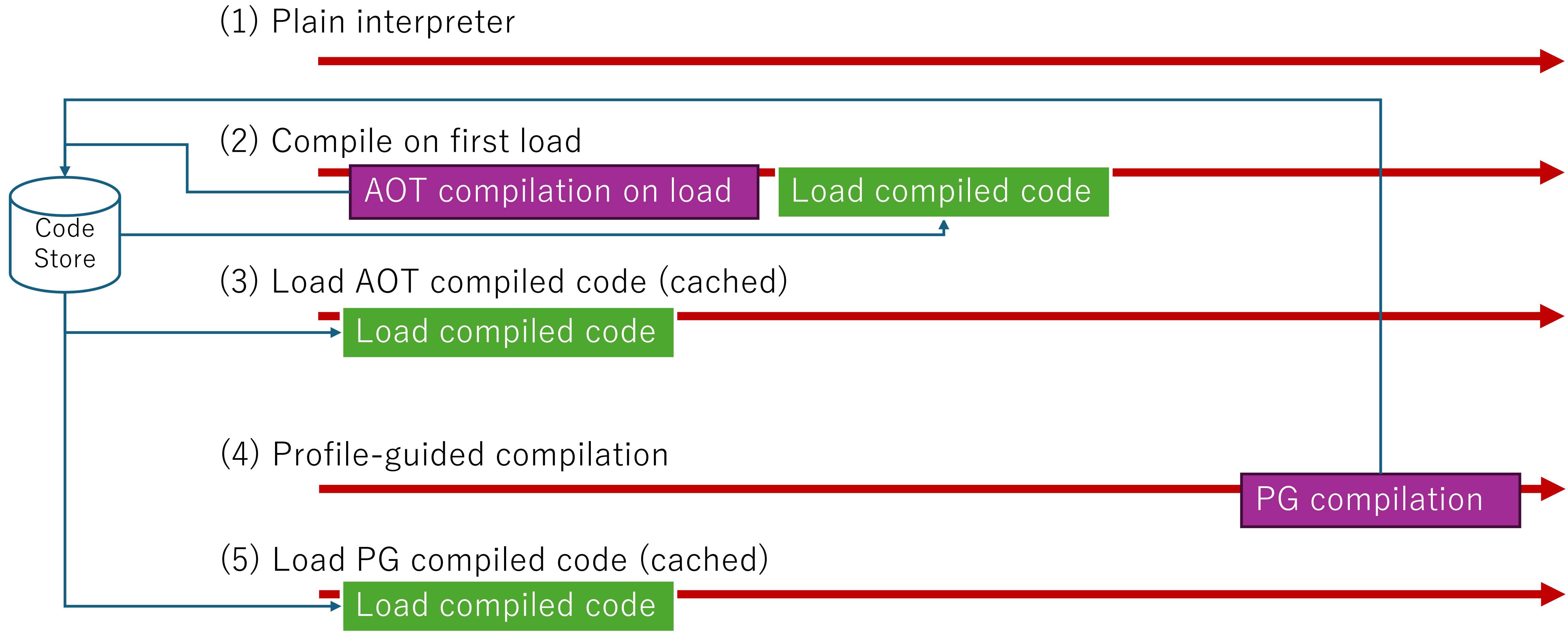
abruby supports 2 compilation types

1. AOT compilation
   — Compile all of AST only with AST structures
2. Profile-guided (PG) compilation
   — Compile using **profile information** collected at run time
   - Which nodes are frequently used? → Compile only N invoked nodes
   - Which types are used for type-specific operation nodes?
   - Which receiver types are used at method dispatch sites?
   - Which shape is used for ivar accesses?
   - Which branch is taken frequently?

We have not yet tried JIT compilation on abruby, but we tried on another interpreter on ASTro
See our paper for details: *ASTro による JIT コンパイラの試作 on PPL2026*

abruby (on ASTro) supports cross-process sharing

Running mode for benchmark

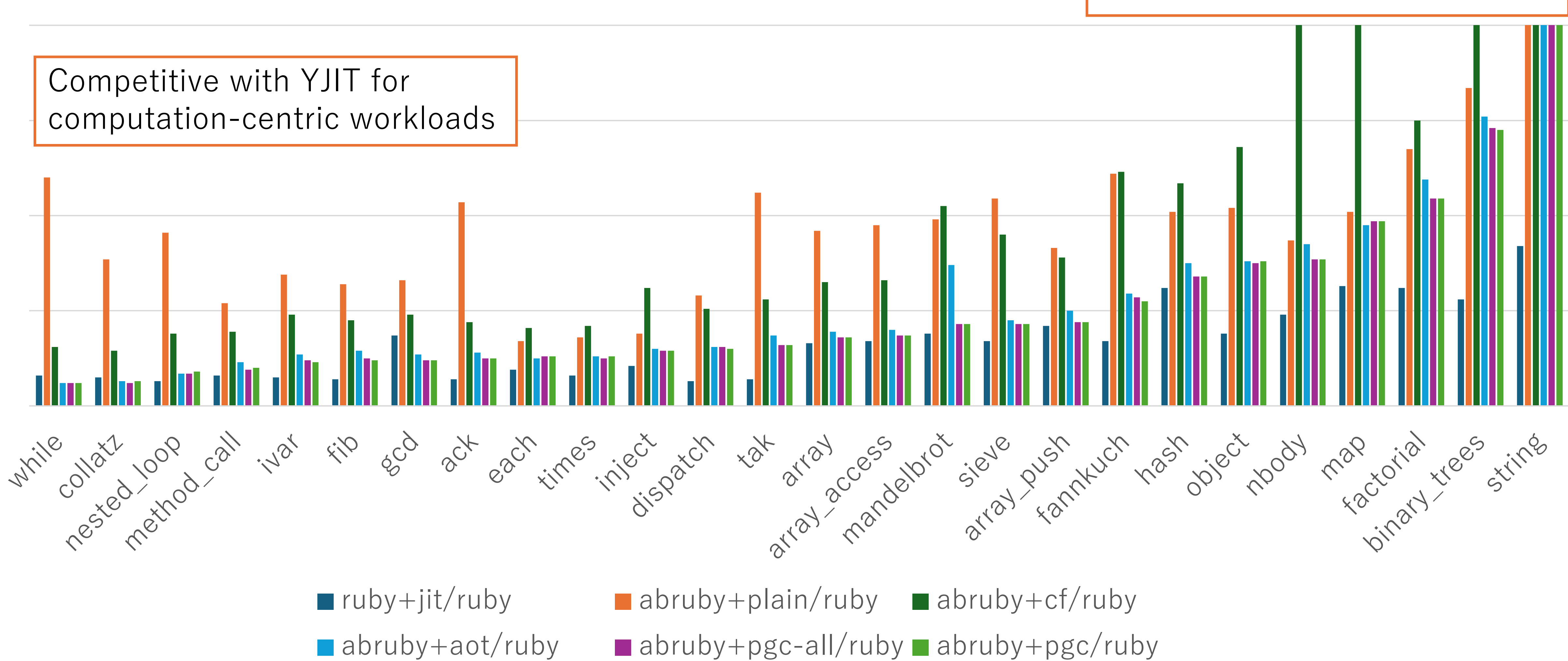


In the common case, abruby loads AOT/PG-compiled code if available in the code store

Microbenchmark results

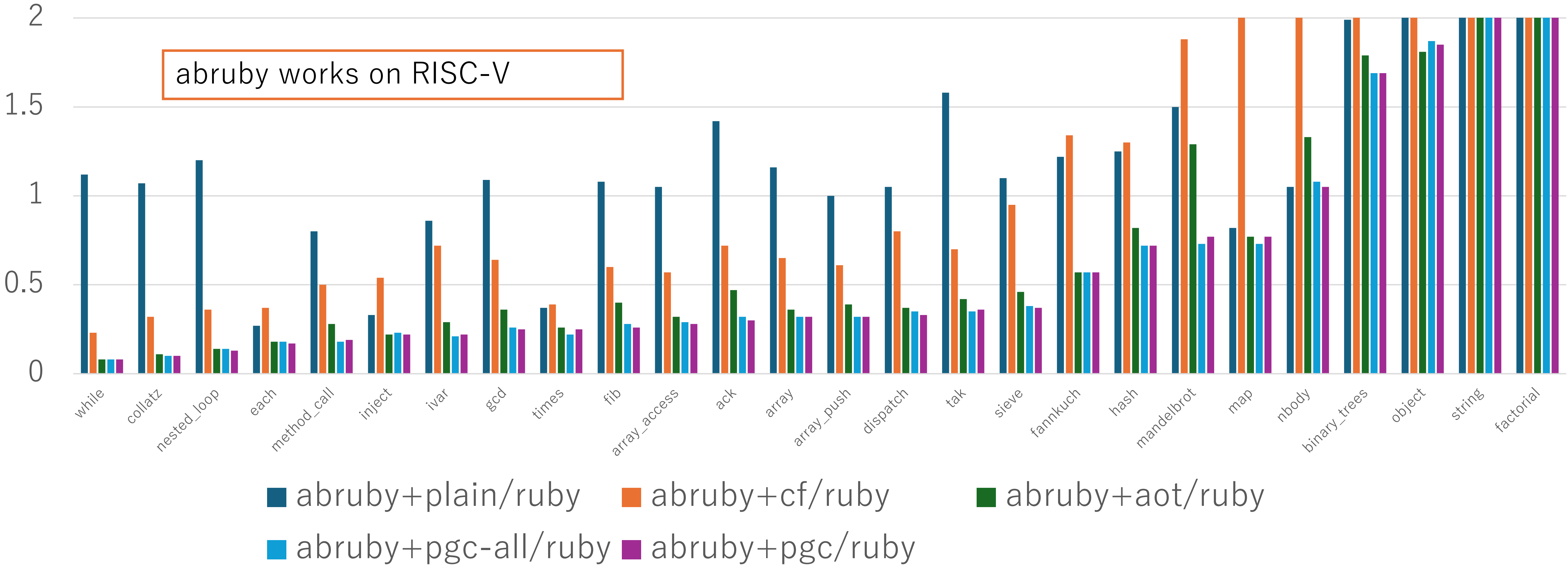
Execution Time (x86_64) Relative to Plain Ruby (Lower Is Better; Capped at 2)

Using Ruby 4.0.2 for baseline



Benchmark results Microbenchmarks (RISC-V)

Using Ruby master for baseline



Using Ruby 4.0.2 for baseline

Benchmark results
Optcarrot (x86_64)

runner	fps	time
ruby	48.74	4.00s
ruby --jit	**206.05**	**1.27s**
abruby --plain	48.90	4.28s
abruby AOT compile first	**83.96**	**23.98s**
abruby AOT (cached)	79.77	2.63s
abruby PG (threshold=0)	45.38	21.60s
abruby PG (threshold=0, cached)	80.51	2.60s
abruby PG (threshold=100)	42.67	10.11s
abruby PG (threshold=100, cached)	**81.04**	**2.59s**

Fastest! (x4)

Fast!
But needs compilation time

Better compilation time
Expected it to be faster…

Future work

- abruby should implement more of its own runtime
  - The CRuby extension approach is easy to implement, but introduces some overhead
- Try a JIT framework on top of ASTro
- Custom native-code loader
  - We currently use dlopen to load compiled code, but cross-process reuse requires our own loader
  - The loader must patch process-specific data into the compiled code
- More performance analysis
  - Find out whether this is a fundamental limit or implementation issue
  - Ruby-level method/block inlining has not been tried yet

AST's strengths

Summary

- A new technique for building fast interpreters –
  ASTro: AST-based Reusable Optimization Framework
  - Generates **fast native code** from a tree-walking interpreter
  - Not limited to Ruby, but Ruby is the primary target
  - Still in the research phase
- Motivations: the growing complexity of Ruby
- How ASTro works
- Early evaluation results

abruby: a bit Ruby on ASTro
  - abruby achieved x1.7 speedup over plain Ruby on optcarrot
  - YJIT is much faster (x4), there is still plenty of room for improvement