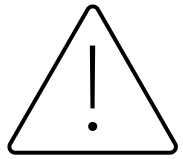
ASTro: An AST-Based Reusable Optimization Framework

Koichi Sasada
STORES, Inc.

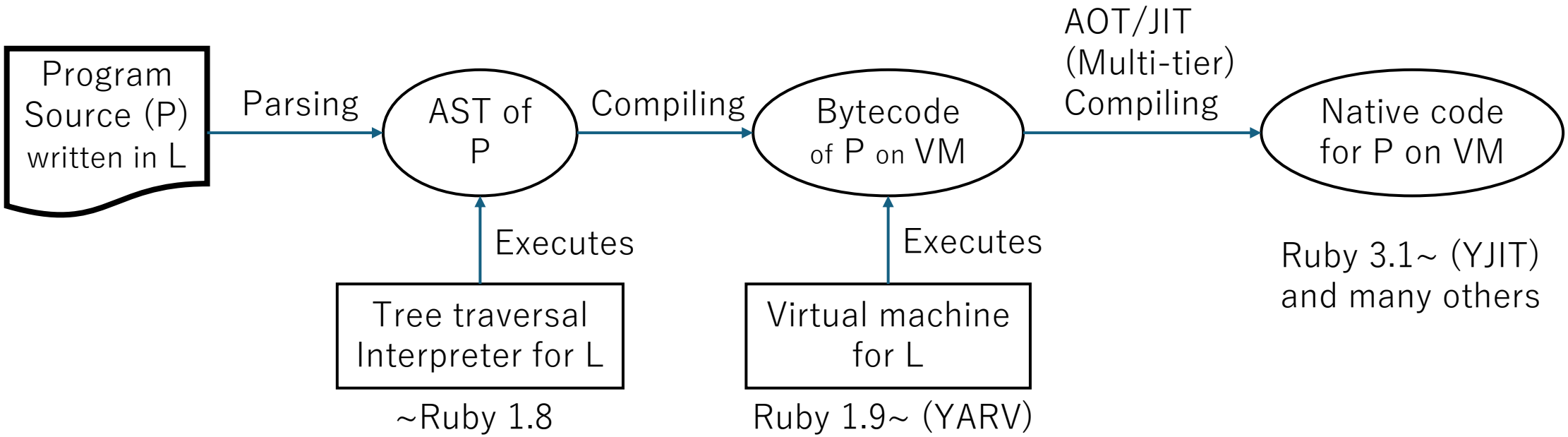


Koichi is one of the Ruby committers and the author of the current Ruby VM (YARV).

About this talk

- Q: How to make an **easy-to-develop** & **fast** interpreter?
- A: ASTro is a lightweight "**Partial evaluation**" (PE) based compiler generator
  - Outsource the native code compilation to C compiler
  - Idea: Specialize the interpreter by generating specialized dispatchers **without complicated transformations**
  - Idea: **Naming AST nodes with "Merkle hash"** to reuse the compiled native code across different processes

Motivation (1) Implementation complexity

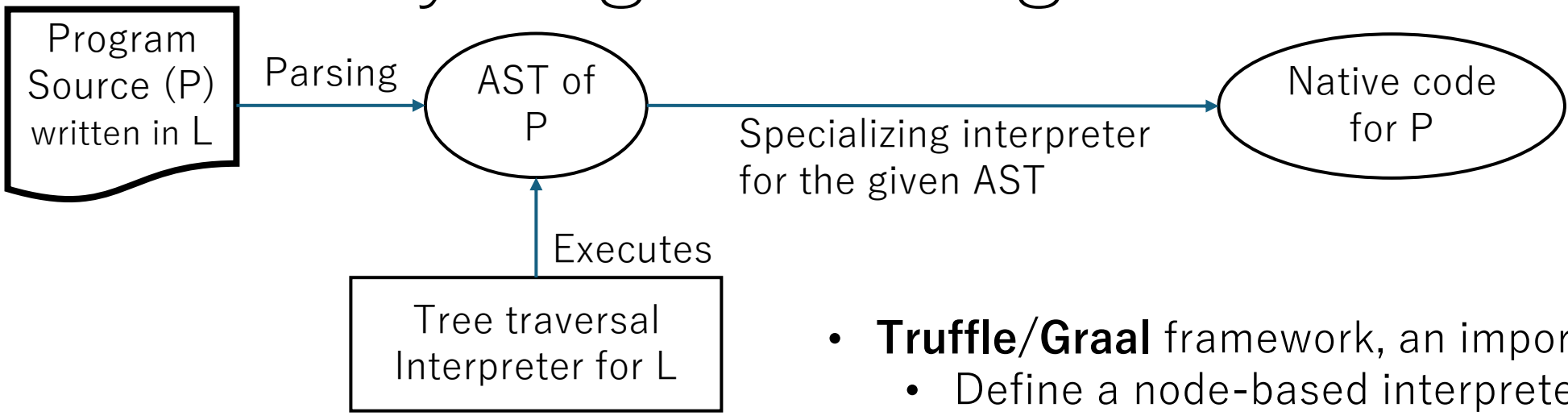


✅ Simple, Easy
❌ Slow

✅ Fast
❌ Complicated

❓ Is it possible to achieve both speed and simplicity?
Disclosure: I dislike assembly programming.

Motivation (2) Partial Evaluation (PE), 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 **Truffle SDL**
  - Profile given node execution
  - Specialize (PE) the given AST and interpreter using **Truffle framework** and generate native code with **Graal compiler**

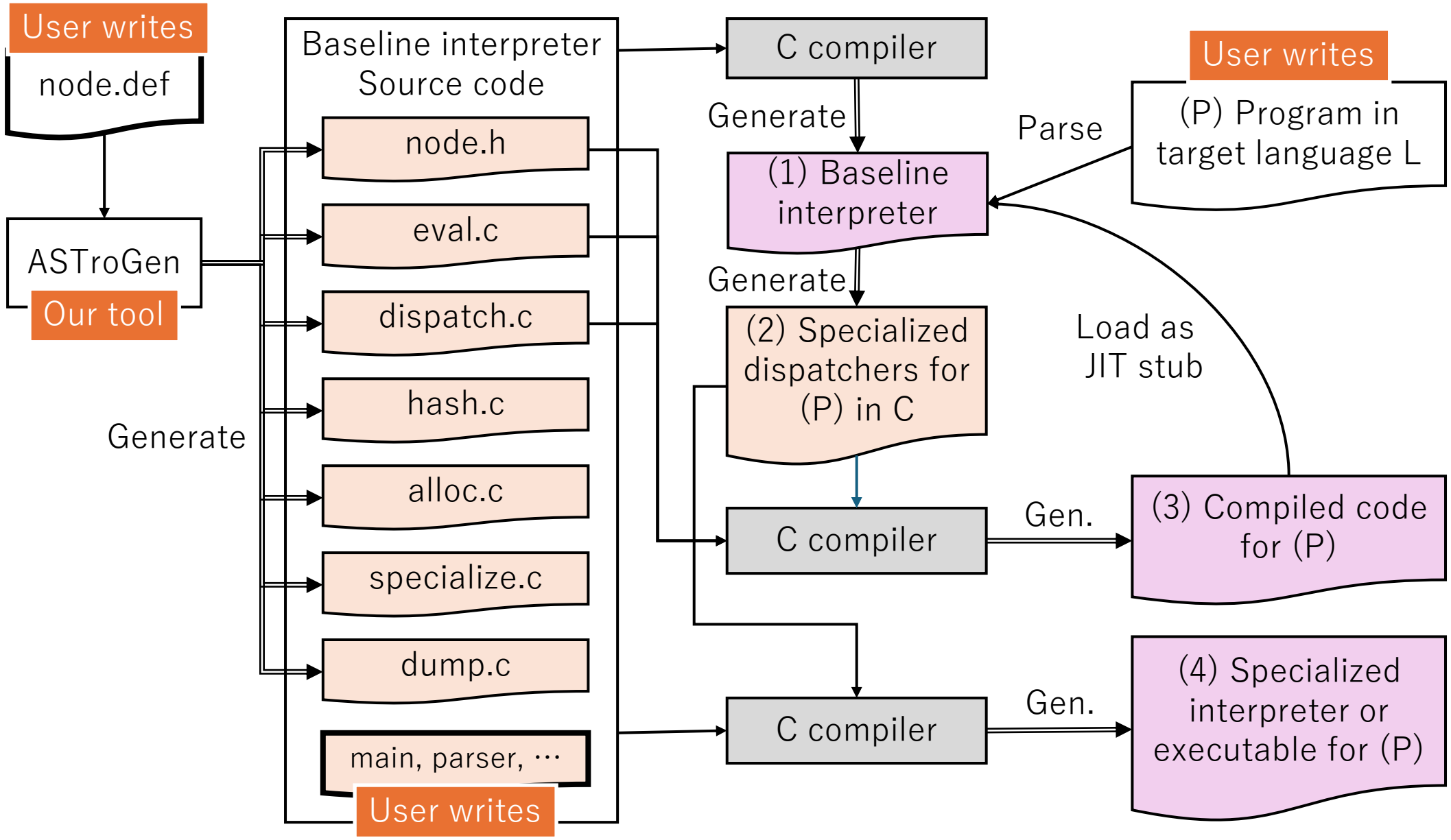
❓ Can we make a lightweight PE framework?

Do you know of a tool that can **generate highly optimized native code** for most of target architectures and is **widely available** and **easy to use**?

Proposal: Lightweight PE framework using a C compiler

- ASTro: AST-based reusable optimization framework
  - Our small tool **ASTroGen** (~500 lines in Ruby) generates **an interpreter** based on user-defined node behavior in C and outsource to C compilers
  - Generates **specializer (PE)** for each node type to make specialized C code
    - **Idea**: Perform PE by generating **only specialized dispatchers** (tiny C snippets), without complicated transformations
  - Commodity C compiler generates **reusable** native code
    - **Idea**: Name the compiled native code with the **AST's Merkle hash**, and **reuse it across runs** to hide long compilation latency

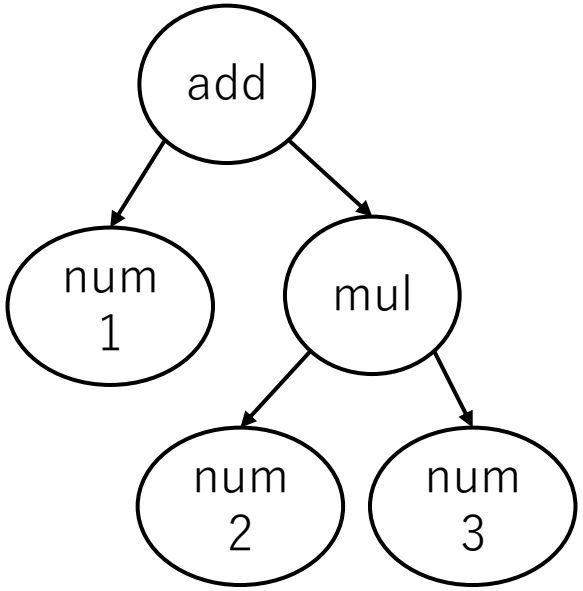
About this talk



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); }
```

```
1 + 2 * 3
```



Node structs

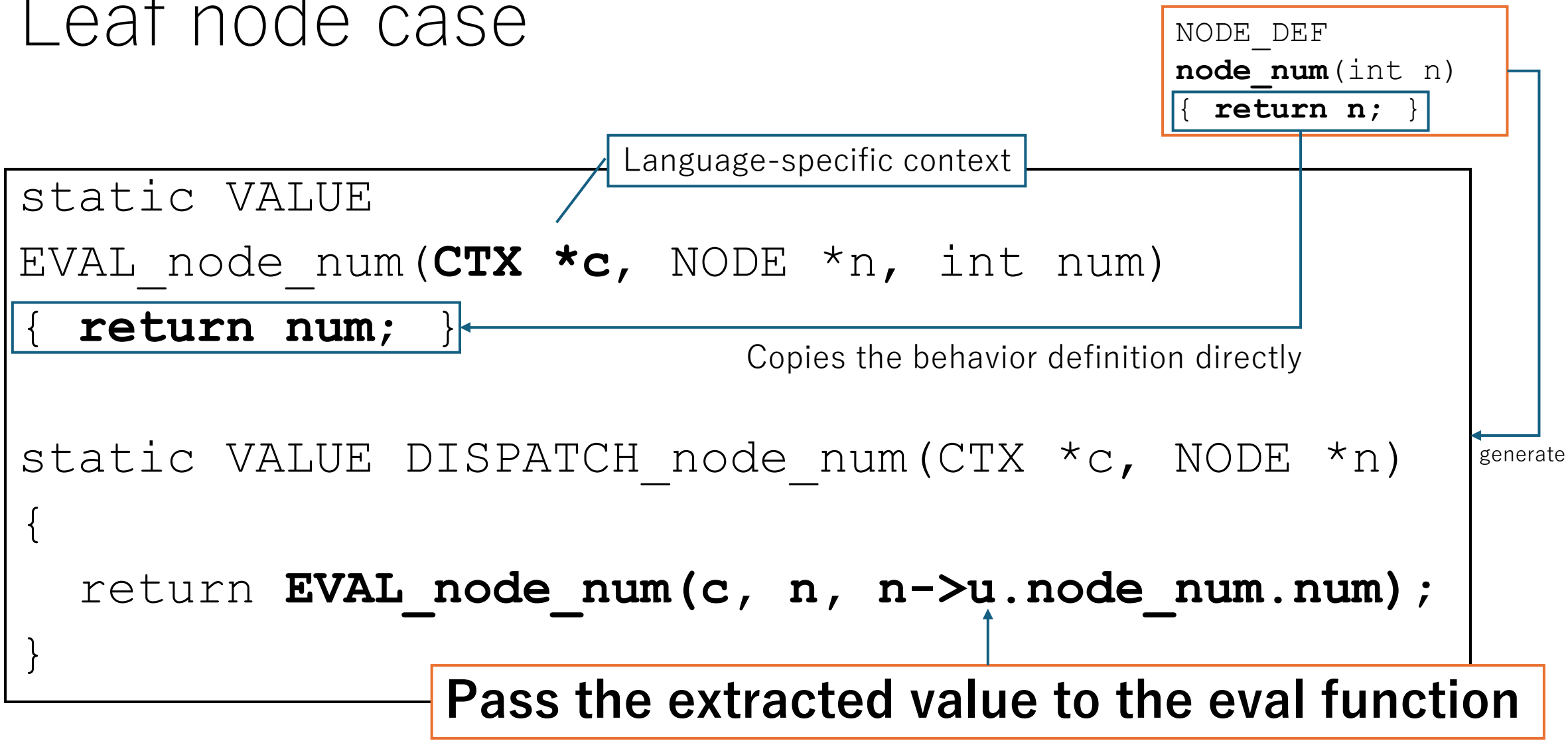
- ASTroGen generates the node structs
  - The Num node has the “int num” field
  - The Add and Mul nodes have lv/rv node-pointer fields
- Each node has a set of functions
  - **Dispatcher**
  - **Specializer**
  - **Hash function**
  - Dumper
- Each node has the name of the dispatcher

```
// node.def
NODE_DEF        // start of the definiton
node_num(int n) // node attributes
{ return n; }   // 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); }
```

generate

```
// created automatically

struct node_num_struct {
  int num;
};
struct node_add_struct {
  NODE * lv;
  NODE * rv;
};
struct node_mul_struct {
  NODE * lv;
  NODE * rv;
};
struct Node {
  struct NodeHead head;
  union {
    struct node_num_struct node_num;
    struct node_add_struct node_add;
    struct node_mul_struct node_mul;
  }u;
};
```

Separating evaluators and dispatchers
Leaf node case



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);
}

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));
}
```

Receives **child node** along with that **dispatchers**, used in the `EVAL_ARG()` macro (simply calls dispatchers)

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

Passes the extracted value with corresponding dispatchers to the eval function

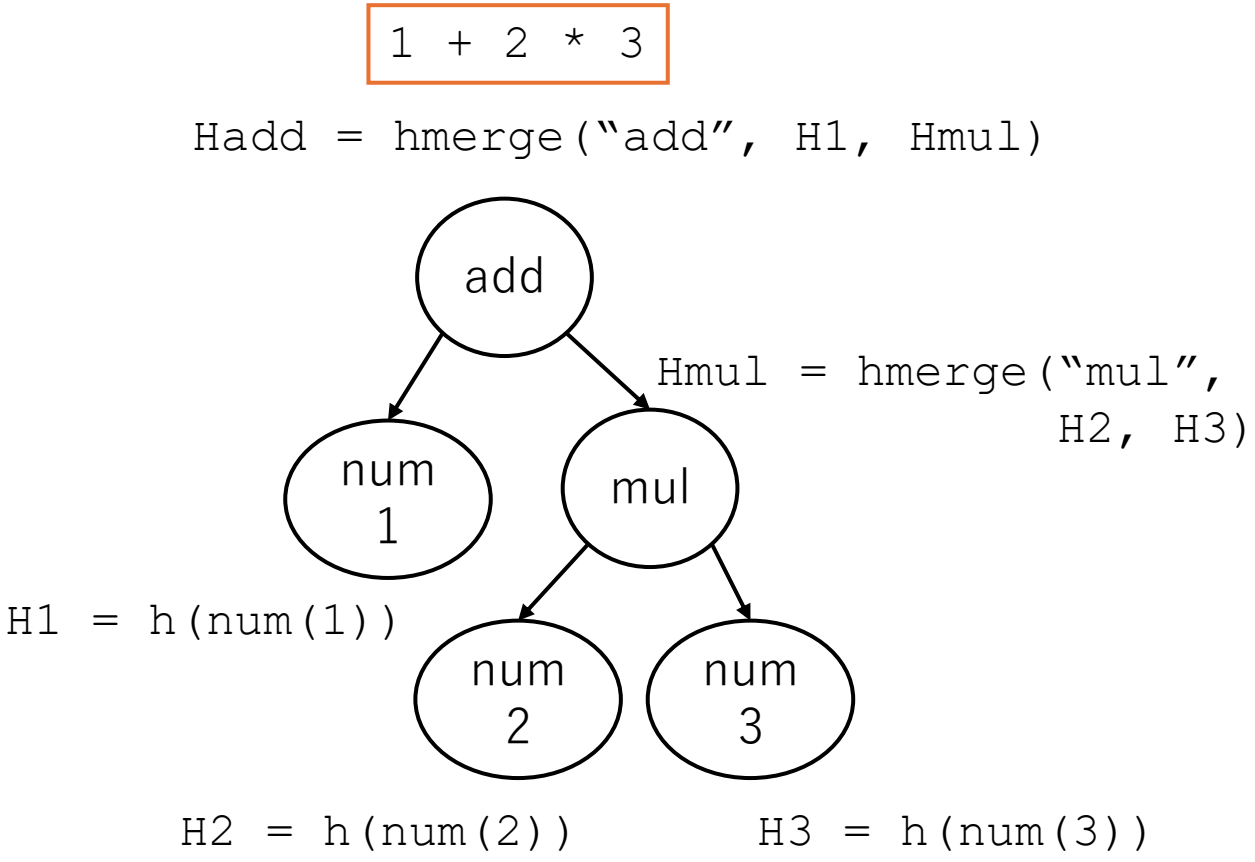
`disp(n)` is a simple macro to get a dispatch function pointer from the node n

Node traversal interpreter from `node.def`

- An interpreter is generated from node.def, and the user provides a parser (with automatically generated allocator functions)
- To evaluate the node “n”, simply calls that dispatcher
  - Each node has its own dispatcher, which calls the corresponding evaluator
  - `#define EVAL(c, n) (*disp(n))(c, n)`
- **Easy transformation** from node.def to an interpreter
- But still a **slow** interpreter yet

Idea 1:
Naming AST node with Merkle-Hash

- Merkle hash
  - Compute each node's hash from its own value and the hashes of its children
- We can uniquely identify any AST subtree (node) structure by its hash value, **even across different interpreter runs**



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

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

Idea 2:
Partial evaluation by generating dispatch functions

- Partial evaluation by generating **ONLY specialized dispatchers** for specific nodes
  - Generates specialized dispatchers for the specific nodes
  - The C compiler can inline the dispatcher function with an eval function

```
// 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

num
2

H2 = h(num(2))

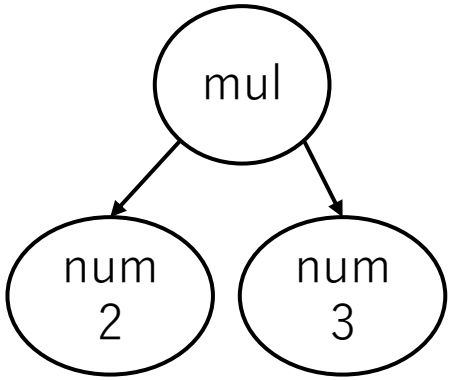
ASTroGen generates

```
// specialize.c

static void
SPECIALIZE_node_num(FILE *fp, NODE *n)
{
  /* name is SD_<hash> */
  const char *name = alloc_dispatcher_name(n);
  n->head.dispatcher_name = name;
  fprintf(fp, "static VALUE¥n");
  fprintf(fp, "%s(CTX *c, NODE *n)¥n{¥n", dname);
  fprintf(fp, " return EVAL_node_num(c, n, %d);¥n",
          n->u.node_num.num);
  fprintf(fp, "}¥n");
}
```

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



Generates specialized code

ASTroGen generates

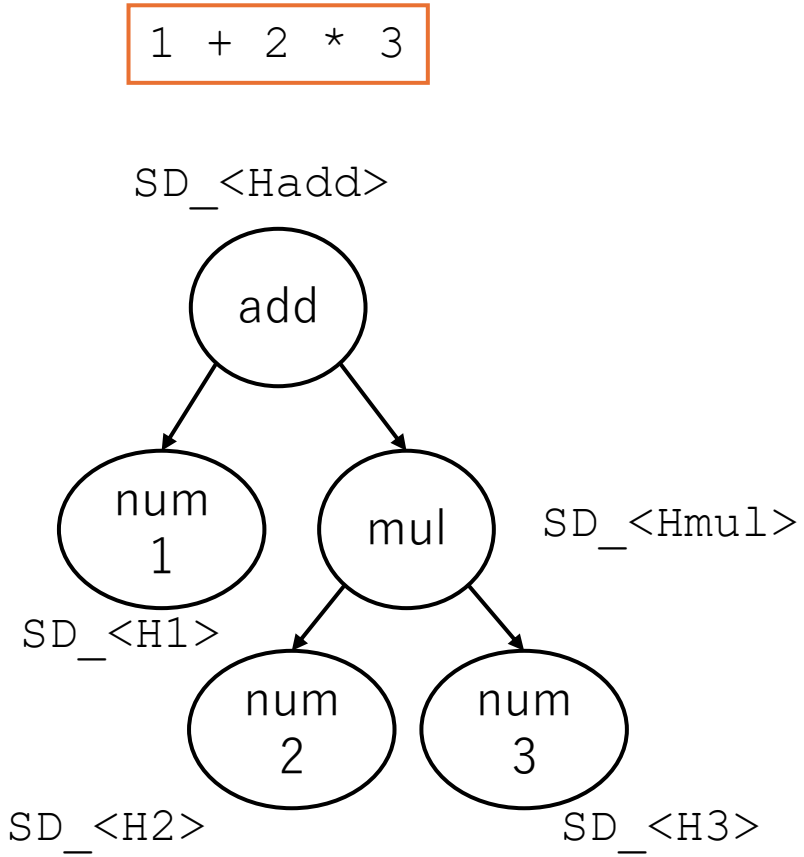
```
// specialize.c

static void
SPECIALIZE_node_mul(FILE *fp, NODE *n)
{
  SPECIALIZE(fp, n->u.node_mul.lv);
  SPECIALIZE(fp, n->u.node_mul.rv);
  const char *dname = alloc_dispatcher_name(n);
  n->head.dispatcher_name = dname;
  fprintf(fp, "static VALUE¥n");
  fprintf(fp, "%s(CTX *c, NODE *n)¥n", dname);
  fprintf(fp, "{ return EVAL_node_mul(c, n,¥n");
  fprintf(fp, " n->u.node_mul.lv, %s,¥n",
              n->u.node_mul.lv->head.dispatcher_name);
  fprintf(fp, " n->u.node_mul.rv, %s);¥n",
              n->u.node_mul.rv->head.dispatcher_name);
  fprintf(fp, " return v;¥n}¥n¥n");
}
```

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

Idea 2:
Partial evaluation by generating dispatch functions

- We can generate all SDs for the AST
- Recent commodity C compilers can inline all dispatchers and evaluators, and **SD_<Hadd>** is compiled with **optimal** machine instructions (just returns 7)
- **SD_<Hadd>** is a **unique identifier** for the AST structure, allowing compiled code to be reused **across processes**

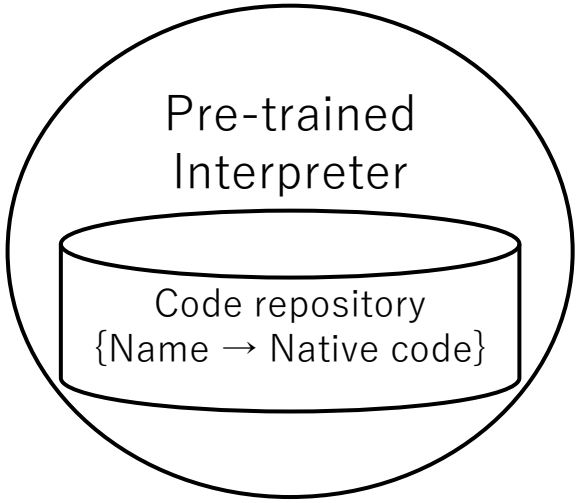
```
0000000000001a20 <SD_<Hadd>>:
1a20: endbr64
1a24: mov $0x7,%eax ; literal 7
1a29: ret
```



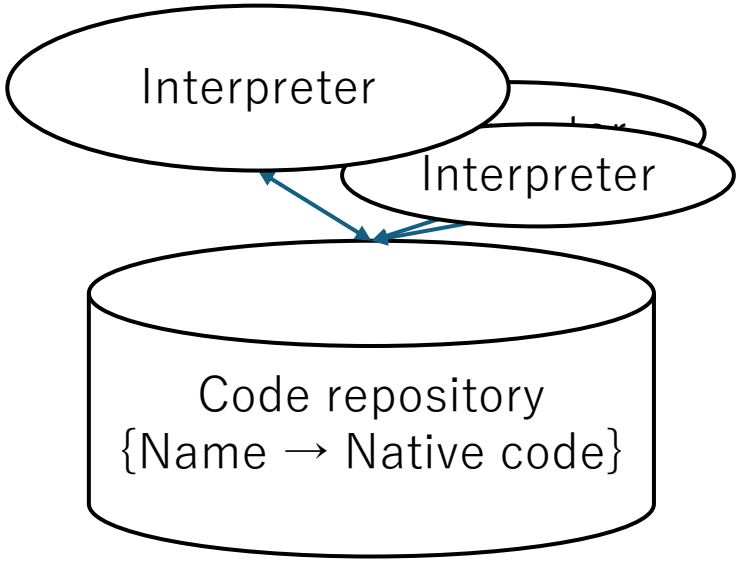
Use cases of our techniques

- Interpreter with simple optimization such as super-instructions
- Compiler or program-specific specialized interpreter
- Profile-guided compiler / interpreter
  - Partial evaluation can utilize runtime profiling information
- JIT/AOT compiler
  - Although C compilation is slow, compiled results can be reused later via Merkle-hash naming.

In process code repository



Out-of-process code repository



Compilation remotely
is also possible

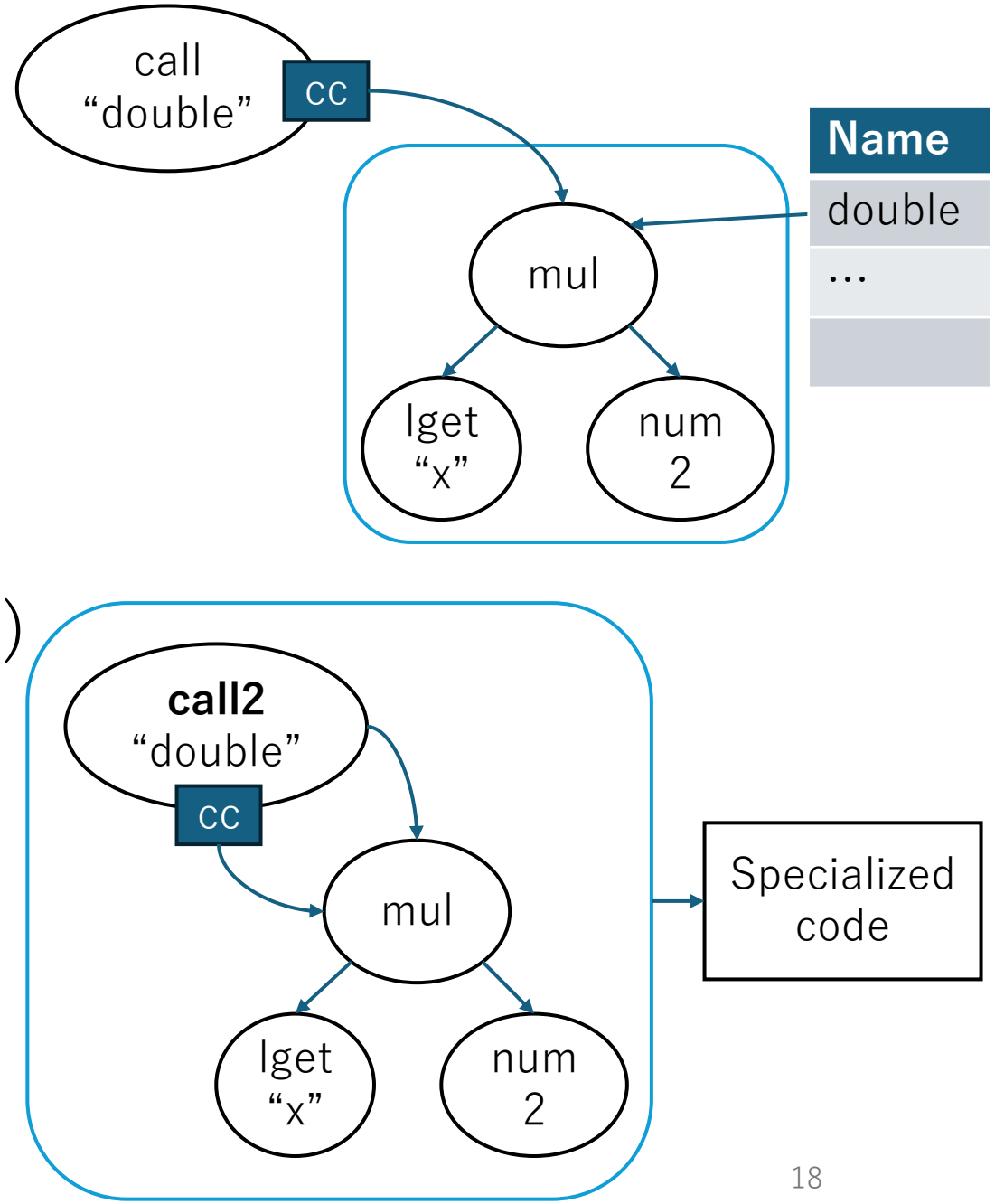
Preliminary evaluation

- naruby: not a Ruby, but a very small subset of Ruby
  - All values are integers (No GC, No OO)
  - No redefinition for binary operators
  - No exception handling
  - All node Types (21 types):
    - Literals: `node_num`
    - Control flow: `node_seq`, `node_if`, `node_while`
    - Local variables: `node_scope`, `node_lget`, `node_lset`
    - Functions: `node_def`, `node_call`, `node_call2`
    - Binary operators: `node_add (+)`, `node_sub (-)`, `node_mul (*)`, `node_div (/)`, `node_mod (%)`, `node_eq (==)`, `node_neq (!=)`, `node_lt (<)`, `node_le (<=)`, `node_gt (>)`, `node_ge (>=)`

```
// a part of node.def
// straightforward definitions

NODE_DEF
node_if(CTX *c, NODE *n, NODE *cond,
        NODE *then_node, NODE *else_node)
{ if (EVAL_ARG(c, cond))
    return EVAL_ARG(c, then_node);
  else
    return EVAL_ARG(c, else_node);
}

NODE_DEF
node_while(CTX *c, NODE * n,
           NODE *cond, NODE *body)
{ while(EVAL_ARG(c, cond))
    EVAL_ARG(c, body);
  return 0;
}
```

naruby: lazy binding call with call cache (CC)

- Lazy binding
  - Looks up a global function table on each call
  - To simulate dynamic language behavior
- Call node has an inline call cache (CC)
  - The CC is invalidated when another function is defined
- “Call2” node merges cached AST
  - PE with cached target function
  - To evaluate profile guided optimization



Evaluation setup

- Benchmarks
  - loop, fib, call, prime_count
- Settings
  - naruby/interpreter: plain interpreter
  - naruby/compiled (AOT) (*)
  - naruby/pg (using **call2** to compile with larger ASTs) (*)
  - gcc/-O0, -O1, -O2 with equivalent C programs (*)

  * Compilation and profiling time (for naruby/pg) is excluded from results
- Architectures
  - x86_64
  - RISC-V (Truffle/Graal cannot generate RISC-V binaries)

Evaluation results (in seconds)

x86_64

		loop	fib	call	prime_count
naruby	naruby/interpret	**0.786**	**4.870**	6.760	**6.170**
	naruby/compiled	**0.001**	**1.093**	**3.435**	**0.444**
	naruby/pg	0.001	**1.143**	**2.061**	**0.443**
C	gcc/-O0	0.042	0.480	1.121	**0.490**
	gcc/-O1	0.023	0.400	1.027	**0.005**
	gcc/-O2	0.001	0.115	0.318	0.434

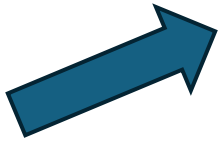
RISC-V

	loop	fib	call	prime_count
naruby/interpret	**8.096**	**45.580**	62.268	**53.177**
naruby/compiled	**0.003**	**6.834**	**17.657**	**1.289**
naruby/pg	0.003	**7.053**	**8.566**	**1.289**
gcc/-O0	0.544	5.481	8.385	**2.476**
gcc/-O1	0.061	1.132	3.151	**0.017**
gcc/-O2	0.002	0.604	1.575	1.280

- In seconds, lower is better
- Measuring while process execution
  - Including AST building, compiled code fetching and execution
  - Excluding native code compiling time

Discussion

- Limitations
  - Hard to achieve peak performance
    - Deoptimizations, exception handling, …
  - Strongly depends on the inlining capability of the C compiler



But EASY!!

- Many open questions and see our paper!!
  - How to store and share the compiled code?
  - How to mitigate code size explosion?
  - How to handle runtime context-aware hashing?
  - How to handle process internal values? → Copy&Patch
  - How to select the hash algorithm?
  - Is it applicable for any languages?
  - Is it possible to use other than C language as a backend?

Related work

- Partial evaluation, meta-circular interpreter
  - pypy, TruffleRuby
- Sharing compiled code between processes
- AST vs. Bytecode
- Code generation
  - Cython, vmgen, …
  - libjit, llvm, …

Conclusion

Check ASTro framework:
https://github.com/ko1/astro/

- Lightweight PE framework: ASTro
  - Performs **partial evaluation** into C code using easy-to-implement approach
    - Specializes the interpreter for a given AST by generating **only dispatchers**
    - Leverages a commodity C compiler to produce efficient native code via inlining
  - Compiled code can be shared across runs by **Merkle-hash-based identity**
- Preliminary “naruby” evaluation
  - Demonstrate promising performance which approaching the performance of gcc/-O0 for equivalent C code
- Future work
  - Extend ASTro to richer languages such as Ruby
  - Explore JIT compilation
  - Investigate remote or distributed code caching