ASTro: An AST-Based Reusable Optimization Framework

Koichi Sasada
ko1@st.inc
STORES, Inc.
Tokyo, Japan

Abstract

Partial evaluation of abstract syntax tree (AST) traversal interpreters removes interpretation overhead while maximizing developer productivity; a language author specifies only the behavior of each AST node, and the framework specializes whole programs automatically. Existing solutions, however, come with heavyweight toolchains and tightly coupled, platform-specific back-ends, making portability and deployment difficult.

We present ASTro, a lightweight framework that keeps the node-centric workflow but eliminates heavy dependencies. ASTro translates the partially evaluated interpreter into well-structured C source code that encourages aggressive inlining by commodity compilers, yielding competitive native code. Because the output is plain C, it can be rebuilt with any mainstream toolchain, reducing deployment effort. To support just-in-time use, every AST sub-tree receives a Merkle-tree hash; identical fragments share their compiled artifacts at astro-scale—across processes, machines, and deployments—so each piece is compiled once and reused many times.

This paper introduces ASTro, a framework for building interpreters and partial evaluators, along with its generator tool, ASTroGen. It shows that language authors can implement interpreters by specifying only the behavior of AST nodes. We present empirical measurements on micro benchmarks that quantify ASTro's runtime performance.

CCS Concepts: • **Software and its engineering → Interpreters**.

Keywords: Interpreter, Compiler, Partial evaluator

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VMIL '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2164-9/25/10
https://doi.org/10.1145/3759548.3763371

ACM Reference Format:
Koichi Sasada. 2025. ASTro: An AST-Based Reusable Optimization Framework. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3759548.3763371

1 Introduction

The simplest way to implement a language runtime is to build an abstract syntax tree (AST) from source code and run a recursive tree-traversal interpreter. However, when performance becomes a priority, implementers usually introduce additional stages: a byte-code compiler, a virtual machine (VM), and ultimately a just-in-time (JIT) compiler that emits native code. While each stage increases speed, it also multiplies maintenance cost: every new grammar feature or optimization must propagate through the entire pipeline, and later stages tightly depend on the exact semantics of earlier ones.

Partial evaluation and meta-tracing compilation offers an attractive alternative[8]: given an interpreter and a program, specialize the interpreter with respect to that program and compile the result. Frameworks like RPython[3, 12] and Truffle/Graal[13–15] demonstrate that this idea can achieve competitive performance, but they rely on heavyweight, tightly coupled toolchains that require significant effort to learn and deploy, for example, when targeting new CPU architectures or constrained environments.

We introduce ASTro, a lightweight and portable partial evaluation framework. A Ruby script of roughly 500 lines, ASTroGen, takes a set of node behaviors and automatically generates (a) an evaluator of tree-traversal interpreter, (b) a partial evaluator that generates a specialized evaluator for the given program, and (c) the required helper routines.

Commodity C compilers then turn specialized code into highly optimized binaries through aggressive inlining, eliminating the need for custom back ends. To reduce compilation latency and enable both ahead-of-time and JIT compilation, we attach Merkle-tree hashes[11] to every AST sub-tree; identical fragments share their compiled artifacts in an astro-scale cache that spans processes, machines, and deployments.

The contributions of this work include the following:

- A method that employs partial evaluation to transform the node behavior into compact C code that commodity compilers inline effortlessly, thus yielding high-performance native code without any complicated intrusion of the node definitions.
- A Merkle-tree hash based scheme for cross-process sharing of compiled AST sub-trees.
- The implementation and open-source release of ASTro framework[1], a compact tool that automates language runtime construction.
- An empirical evaluation on micro benchmarks quantifying ASTro's runtime performance.

2 Background

This section reviews the technical context in which ASTro is positioned.

2.1 Tree-Traversal Interpreters

The most straightforward runtime organizes execution as a recursive walk over an abstract syntax tree and executes a per-node evaluation logic. Although this pattern is easy to implement and debug, it tends to perform poorly on modern processors: each step follows pointers to nodes that are scattered in memory, so the traversal touches non-contiguous locations and defeats data-cache locality.

2.2 Byte-code Virtual Machines and JIT Compilers

To mitigate these costs, many language runtimes lower the AST into a linear sequence of byte-code instructions executed by a hand-written VM. This improves branch prediction and locality, yet introduces new maintenance layers:

- An AST-to-byte-code compiler
- A VM instruction set definition and implementation
- Optionally, a JIT back-end that lowers VM instructions to machine code

Each layer depends tightly on the previous one, so adding a grammar feature or changing an opcode semantics requires cascading modifications. JIT compilers further require IR optimizers and CPU-specific code generators, increasing the complexity of maintenance and portability.

2.3 Partial Evaluation

Partial evaluation (PE) specializes a program p with respect to a subset of its input s, producing a residual program p_s that satisfies $p_s(d) = p(s, d)$, where d denotes the remaining input. When p is an interpreter *Int* for the language L and s is a program P_L written in L, the *first Futamura projection* yields a compiled version P_L^*:

$$P_L^* \;=\; \mathrm{PE}(\mathit{Int}, P_L).$$

That is, P_L^* runs with no further interpretation overhead yet preserves the semantics of P_L.

2.4 Existing Frameworks

This section reviews two prominent systems that elevate meta-compilation techniques such as partial evaluation and meta-tracing JIT to a first–class optimization strategy, demonstrating that *interpreter-only development* can indeed deliver high performance—but at the cost of significant infrastructure complexity.

RPython / PyPy. An interpreter written in *RPython* is translated via a whole-program type inference and control-flow analysis that infer low-level types and flow facts. The resulting low-level IR is processed by a translation pipeline that performs type specialization, inserts garbage collection and exception-handling machinery, and finally generates a large C code base. An external C compiler then links the runtime, garbage collector, and application into a monolithic executable. At runtime a meta-tracing JIT observes hot loops, specializes them for the concrete runtime types, and produces optimized machine code.

Truffle / Graal. An interpreter is expressed as a set of node executors written in Java with the Truffle DSL. During program execution in the interpreter, each node profiles the runtime behavior. Once the system detects a hot path, the partial evaluator is applied to the AST with the collected profile data, producing an optimized intermediate representation. The Graal compiler then compiles this IR into native code with deoptimization support. This design enables high-performance native code generation solely from interpreter definitions.

Common Pattern—and Shared Pain. Both frameworks uphold the attractive ideal of keeping a single interpreter as the canonical definition of language semantics while relying on meta-compilation to erase interpretation overhead.

Yet they share one practical obstacle: heavyweight toolchains. Heavyweight toolchains present challenges of poor deployability, limited extensibility, and difficult debugging. For example, porting Truffle/Graal beyond JVM environments or to a new CPU architecture often entails disentangling a highly complex system.

3 ASTro Framework

Existing meta-compilation systems demonstrate that "an interpreter alone can run fast," yet they remain daunting in practice: their toolchains are large and difficult to use, so bringing one into a project—or porting it to a new CPU architecture—requires deep expertise and substantial engineering effort.

To address these challenges, ASTro adopts the following ideas:

[1] https://github.com/ko1/astro

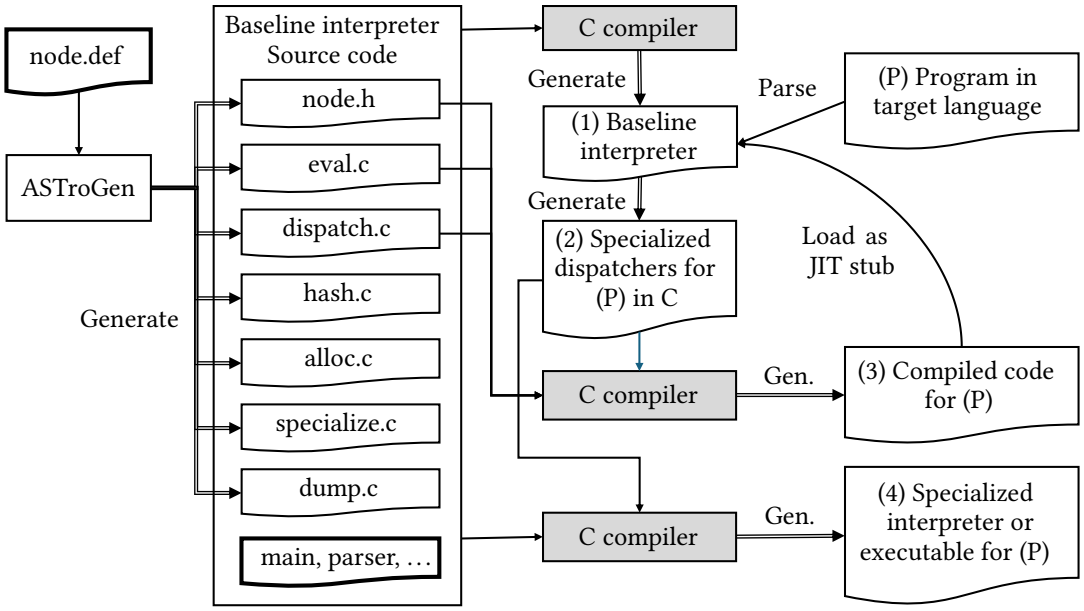

Figure 1. Workflow with ASTroGen

- **Outsource the hard part.** The user only needs to write the behavior of each node in the well-known C language, while all low-level optimizations and code generation are delegated to a commodity C compiler that is widely used, well tested, highly optimizing and expected to remain well maintained in the future.
- **Inline-friendly specialized output.** The partial evaluator generates small dispatchers that leave the evaluator bodies untouched, making them trivial for commodity C compilers to inline and optimize.
- **Hash-based reuse.** Each sub-tree is named by a Merkle-tree hash of its structure. Identical hashes imply identical semantics, which allows any interpreter process reuse a previously compiled code without repeating compilation.

This section explains ASTro's overall design and key ideas with examples.

3.1 Workflow Overview

As illustrated in Figure 1, ASTro takes a node definition file (`node.def`) and generates parts of the interpreter codebase in C. It produces `node.h` for AST structure definitions, `eval.c` for evaluators, `dispatch.c` for evaluator dispatchers, `hash.c` for Merkle hashing functions, `alloc.c` for node constructors, `specialize.c` for partial evaluators, and `dump.c` for human-readable node printers. Each file contains per-node functions systematically derived from the node definitions, ensuring consistency and extensibility across the interpreter.

The user constructs an AST using their own parser and the allocators provided in `alloc.c`, then executes the program

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

Figure 2. `node.def` of the toy calc language

by invoking `EVAL(c, ast)` with a user-defined context `c`. This setup yields the baseline interpreter ((1) in Figure 1).

At any point, the interpreter may use the partial evaluator to emit node-specific dispatchers in C ((2) in fig.). It can then invoke a C compiler to generate native code on the fly ((3) in fig.) and load it as a JIT stub. Since each generated symbol is named using a Merkle-tree hash, identical sub-trees can be reused across processes. The user may also produce a specialized (pre-trained) interpreter ((4) in fig.) with specialized dispatchers for a given program P, which can be treated as a standalone executable for P.

3.2 Sample Scenario

To demonstrate the workflow and illustrate ASTro's techniques, we present a toy calc language comprising only numbers, `+`, and `*`. Its `node.def` lists three nodes: number literal,

addition, and multiplication, shown in Figure 2. Each node declares its fields as C-style parameters and specifies its behavior in plain C; child node evaluation is expressed via the macro `EVAL_ARG()`.

ASTroGen generates an evaluator from the `node.def` file. A working interpreter can then be constructed by combining it with a user-supplied, language-specific parser and context management.

3.3 Declarations

ASTroGen derives the complete `NODE` structure definitions directly from `node.def`; see Listing A for the emitted code. This eliminates the need to manually define enumerations or maintain node-kind registries.

3.4 Node Evaluators

In AST traversal interpreters implemented in C, a `switch`/`case` statement typically dispatches on the syntactic tag of each AST node. ASTro, by contrast, embeds in every node header a pointer to its own evaluator function that evaluates that particular node instance—mirroring the role of Truffle's `execute()` method.

Per-node evaluator. With given `node.def` definition, for a numeric literal we generate an evaluator `EVAL_node_num` that simply returns its embedded value; the first argument `CTX *c` is a user-defined runtime context that can hold global variables, a value stack, or any language-specific state:

```
static VALUE
EVAL_node_num(CTX *c, NODE *n, int32_t num)
{
  return num;
}
```

Dispatcher. The evaluator is *not* invoked directly. Instead each node owns a thin dispatcher, `DISPATCH_...`, which extracts the node's fields and forwards them to the evaluator:

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

`n->head.dispatcher` is the address of the dispatcher for this node. A general evaluation macro `EVAL(c, n)` for node `n` can be defined with `(*n->head.dispatcher)(c, n)`.

Nodes with children. When a node owns child nodes, its generated evaluator receives both the child node (`NODE *`) and the child's dispatcher:

```
static 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);
}
```

In this case, this function receives not only `lv`, but also `lv_disp`, the dispatcher of the `lv`.

Here `EVAL_ARG(c, n)` is a macro that expands to the corresponding dispatcher call supplied as an argument[2].

The dispatcher for the addition node therefore collects the two child dispatchers and passes them along:

```
static 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);
}
```

Here `disp(n)` is a macro to get the dispatcher of `n`.

Separating dispatchers from evaluators is a key design choice that makes building a partial evaluator remarkably easy with off-the-shelf C compilers, as we demonstrate in subsection 3.7.

3.5 Computing Node Hashes

To enable cross-process caching and de-duplication, ASTro assigns a Merkle tree hash to every AST node—effectively naming the entire sub-tree rooted at that node. A node's hash is obtained by hashing (i) the node's kind and (ii) each of its attributes.

```
// Leaf node (numeric literal)
static node_hash_t HASH_node_num(NODE *n)
{
  node_hash_t h = hash_cstr("node_num");
  h = hash_merge(h,
      hash_uint32((uint32_t)n->u.node_num.num));
  return h;
}

// Internal node (addition)
static node_hash_t HASH_node_add(NODE *n)
{
  node_hash_t h = hash_cstr("node_add");
  h = hash_merge(h, hash_node(n->u.node_add.lv));
  h = hash_merge(h, hash_node(n->u.node_add.rv));
  return h;
}
```

The latter is a *Merkle-tree* construction: the hash of a parent re-uses the already computed hashes of its children, ensuring that two identical sub-trees—regardless of where they appear—share the same digest. Because ASTro names specialized dispatchers after this digest, any process can look up machine code produced earlier.

ASTro ships default hashers for the common field types such as integers, C strings, and child nodes, so users need to supply custom functions only when a node stores non-standard fields (e.g. pointers to user-defined `structs`).

[2] `EVAL_ARG(c, n)` is simply defined with `(*n##_disp)(c, n)`

3.6 Utility Functions

ASTroGen automatically emits two auxiliary families of routines that facilitate front-end integration and debugging.

Allocators (`ALLOC_*`). For each node type the generator produces an allocator, such as `ALLOC_node_num(int32_t n)`, that calls a user-supplied function[3] to obtain memory and then initializes the node's header as well as its fields. A parser generated by parser generators, or any hand-written front end, can therefore construct the AST simply by invoking the appropriate `ALLOC_*` helpers, without direct knowledge of the node layout.

Dumpers (`DUMP_*`). Every node also has a printer that writes a human-readable representation of the node's kind and attributes. These dumpers are indispensable when debugging specialization failures or verifying that distinct syntax trees indeed share the same Merkle hash. For built-in fields (integers, C strings, child node pointers) ASTro provides default formatters; if a node stores application-specific data (e.g., a pointer to a user-defined `struct`), the implementer needs to supply a custom dumper for it.

3.7 Partial Evaluator

Each node type comes with a `SPECIALIZE_*` routine that generates a specialized dispatcher in C. Given a concrete node instance, the routine prints a specialized `DISPATCH` function whose arguments are hard-wired constants; it never inspects, let alone rewrites, the underlying `EVAL_*` implementation. A commodity C compiler can therefore inline the tiny dispatcher together with the generic evaluator, removing every layer of indirection.

Leaf node (numeric literal).

```
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");
}
```

A node such as `node_num(1)` thus yields a dispatcher that does nothing but call `EVAL_node_num(c, n, 1)`.

Specialized function names always begin with `SD_` (short for "Specialised Dispatcher"), followed by the node's Merkle hash—for example `SD_dfb75fdabb0d5ef6`.

Internal node (addition).

The specialising routine for `node_add` first recurses on its children, then emits a dispatcher that delegates to the already specialized children:

```
static void
SPECIALIZE_node_add(FILE *fp, NODE *n)
{ SPECIALIZE(fp, n->u.node_add.lv);
  SPECIALIZE(fp, n->u.node_add.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_add(c, n,\n");
  fprintf(fp, "    n->u.node_add.lv, %s,\n",
        n->u.node_add.lv->head.dispatcher_name);
  fprintf(fp, "    n->u.node_add.rv, %s);\n",
        n->u.node_add.rv->head.dispatcher_name);
  fprintf(fp, "}\n\n");
}
```

Here, each child's `dispatcher_name` is recorded while specializing them, then passed to the parent's evaluator.

For example, when `SPECIALIZE_node_add()` is invoked on a actual node, it prints the dispatcher like below:

```
static VALUE
SD_dfb75fdabb0d5ef6(CTX *c,  NODE *n)
{
    return EVAL_node_add(c, n,
      n->u.node_add.lv, SD_ef2d3a2c467c98a6,
      n->u.node_add.rv, SD_fa5c4f2645bc412
    );
}
```

Inlining by C compilers.

Because the generated code embeds concrete dispatcher function pointers, commodity C compilers can resolve these calls at compile time and inline the referenced functions aggressively, effectively removing the dispatch overhead.

In other words, a partial evaluator can be constructed merely by generating dispatcher functions, producing C code whose structure directly reflects the AST and can be recognized and optimized by commodity C compilers.

Specializing the following AST

```
// 1 + 2 * 3
node_add(node_num(1),
         node_mul(node_num(2),
                  node_num(3)))
```

produces five specialized dispatchers in total; with the gcc version 13.3.0 (x86_64) the outermost one compiles to:

```
0000000000001a20 <SD_dfb75fdabb0d5ef6>:
    1a20: endbr64
    1a24: mov    $0x7,%eax   ; literal 7
    1a29: ret
```

[3] User needs to provide `node_allocate(size_t size)` for allocators.

As we can see, all nodes were successfully inlined, and the resulting native code simply returns the final value 7.

Any future execution that encounters a sub-tree with the same hash can simply locate `SD_dfb75fdabb0d5ef6` in the cache and jump to it, bypassing interpretation entirely.

3.8 When and How to Exploit Specialized Code

The partial evaluator turns an AST into specialized C code that a commodity compiler translates to efficient native code. How that capability is exploited depends on the desired workflow; ASTro supports four canonical scenarios.

1. Plain interpreter. When the partial evaluator is left off, ASTroGen generates a straightforward tree-traversal interpreter. The ability to obtain such an engine from nothing but a single `node.def` file—parsing logic aside—greatly accelerates early prototyping.

Even in this "pure interpreter" mode, developers are free to pre-specialize a small library of high-frequency sub-trees before shipping. Typical examples include nodes that return the constant 0, load the first local variable, or store 0 into the second local variable—where the last is typically implemented as a compound of multiple nodes. Since each pre-specialized fragment is named by its Merkle hash, the interpreter can embed pre-specialized binaries and accelerate hot micro-patterns without invoking the full JIT machinery.

2. Ahead-of-time (AOT) compiler and specialized interpreter. The front end parses the source program and feeds the resulting AST directly to the partial evaluator, producing specialized code without ever executing the interpreter. The resulting code can be used as a specialized interpreter or a standalone executable for the given source program. This yields performance that exceeds interpreter speed, all from just a `node.def` description.

If the language's call targets are statically resolvable, whole-program inlining may even collapse the entire application into a single monolithic function.

3. Profile-guided AOT compiler and interpreter. Programs executed once are often executed again. At interpreter shutdown ASTro can specialize every sub-tree that was actually executed, compile them and store them with the Merkle hash. On the next run, the interpreter can reuse them for the same ASTs and otherwise fall back to normal interpretation. Profile data—such as inline caches of method lookup in a dynamic language—can be recorded on each node and fed back into the specializer, allowing it to embed guarded predictions of future call targets.

4. Just-in-time (JIT) compiler. While the program is running, hot paths can be identified, specialized into C with our partial evaluator, compiled, and dynamically loaded. Although invoking a full C compiler is expensive, ASTro mitigates the cost by hashing: once a hot fragment has been compiled by any process—on the same machine, another node in the cluster, or even an off-planet data center—it can be reused simply by looking up its hash.

In short, our specialization mechanism underlies an AOT compiler, a conventional interpreter, a profile-guided hybrid, and a JIT engine, with the Merkle-tree hash acting as the unifying abstraction for code reuse across runs.

4 Evaluation

To assess both the implementation effort and the runtime performance of ASTro, we implemented *naruby*, a deliberately minimal subset of the Ruby programming language.[4]

4.1 naruby: Not A Ruby

The name "naruby" abbreviates *Not A Ruby*: the subset is intentionally minimal and eschews Ruby's object system entirely. The only runtime value type is a signed integer.

Context and local variables. The runtime context comprises a value stack whose top segment serves as the fixed-size frame for the current function, thereby holding all local variables. `node_lget` and `node_lset` are provided to access the local variables, and `node_scope` is provided to prepare the toplevel frame.

The global function table described below is stored in the same context structure.

Control flow. naruby recognizes exactly three control flow constructs: (1) a *sequence* (`node_seq`) that evaluates its children from left to right, (2) an *if-statement* (`node_if`), and (3) a *while-statement* (`node_while`).

Function definition and call. A `node_def` appends a triple (*name*, *arity*, *AST*) to the global function table, storing the function's identifier, the number of parameters it expects, and a pointer to its body. If a function with the same name is already registered, the new definition overrides the previous one.

At call time the callee is first located by a linear `strcmp` search with the function name over the global function table. The result of this search is then written into an inline cache stored in the `node_call` node. The execution context maintains a monotonically increasing *function-table version*; whenever the table is updated, the version is bumped. The cache records the version number at which it was filled, and on subsequent executions the call site reuses the cached AST pointer whenever the stored version matches the current one, thereby bypassing the `strcmp` scan entirely. Finally, arguments are pushed onto the topmost slots of the value stack before control transfers to the callee.

To expose call-site cache information to the specializer, we added a `node_call2` node which has a `cached_callee` field that stores the dispatcher found by the inline cache at run time. When a call-site sees the same target repeatedly (a

[4] Ruby was chosen simply because the author was familiar with extracting its AST; no other reason influenced this choice.

cache hit), logic in the interpreter switches to `node_call2` and a `cached_callee` to invoke the stored pointer directly. Because the dispatch target is now a concrete function pointer, the partial evaluator can treat it as a constant and inline the call into the caller's code during specialization.

Built-in arithmetic. The basic arithmetic operations are realized as dedicated AST nodes rather than as library calls.

All Node Types (21 types):

- **Literals**: `node_num` (integer constant)
- **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` (>=)

Front-end construction. We reuse the standard Ruby parser to obtain a full Ruby AST and then translate it node-for-node into naruby's AST with `ALLOC_*` functions. Any Ruby construct that lacks a naruby counterpart results in a compile-time error.

4.2 Authoring Effort

The entire `node.def` for naruby is about 300 lines of code. ASTro generates a baseline interpreter from it with a partial evaluator. To make a usable interpreter, we also need a parser to generate the AST, a command-line option parser, a garbage collector and so on.

Implementing the semantics of each node is straightforward and declarative; for example, the `node_if` construct can be defined as follows:

```
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);
}
```

A hand-written source-to-C transpiler is a conceivable alternative but more complex than a tree-walking interpreter. It must translate all control flow into valid C, even though such constructs often have no direct counterpart. By contrast, ASTro requires only per-node descriptions of semantics in C, making the implementation far more straightforward.

4.3 Performance

Performance was evaluated using four micro benchmarks written in naruby:

- **loop**: 100 million iterations of an empty `while` loop.
- **fib**: naive recursive computation the 40th Fibonacci number.
- **call**: one million chained calls through a ten-function pipeline (`def f0(n)=f1(n)`, ..., `def f9(n)=n`)[5].
- **prime_count**: naive counting of all primes ≤ 100 000, repeated 100 times.

Table 1. Micro benchmark results on x86_64 in seconds

	loop	fib	call	prime_count
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
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

Table 2. Micro benchmark results on RISC-V in seconds

	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

All benchmarks measure wall-clock time, from process startup to termination (including parsing and loading compiled code), and report the median of three runs.

We evaluate the following configurations:

- **naruby/interpret** executes via the pure AST traversal interpreter.
- **naruby/compiled** uses ahead-of-time (AOT) compilation of the specialized C output. Compilation time is not included.
- **naruby/pg** uses profile-guided compilation of the specialized C output. Profiling and compilation time are not included.
- **gcc/-O0 to -O2** are the same benchmark programs ported to C and compiled with GCC optimization levels `-O0`, `-O1`, and `-O2`[6]. Compilation time is not included.

All naruby binaries are compiled with GCC (`-O3`). The experiments were carried out on two target platforms:

- **x86_64**: AMD Ryzen 9 5900HX, Ubuntu 24.04, GCC 13.3.0
- **RISC-V**: lpi4a board, Linux 5.10, GCC 13.2.0

[5] In the C-ported version, each $f_n()$ is placed in a separate source file to prevent inlining.

[6] We omit `gcc/-O3` results because they did not differ from `-O2` on these benchmarks.

Because ASTro generates C code, we can easily run naruby on a RISC-V machine.

Table 1 and 2 show the micro benchmark results. The `naruby/compiled` achieves dramatic speedups over the interpreter baseline, closing in on—or even surpassing— `gcc/-O0` in arithmetic intensive benchmarks. In the *loop* benchmark, the C compiler removes the loop entirely for specialized code from our partial evaluator, driving execution time to near-zero. *prime_count* on RISC–V run roughly twice as fast as `gcc/-O0`[7]. In the *call* benchmark, profile-guided compilation (`naruby/pg`) further reduces call overhead, though it still does not quite match `gcc/-O0`—an issue we plan to investigate.

Overall, the fact that a AST-traversal interpreter, specified solely via `node.def`, can approach `gcc/-O0` performance is a remarkable validation of the ASTro framework.

4.4 Generated File and Compilation Time

To evaluate code generation and compilation overhead, we constructed a synthetic input file consisting of 10,000 lines of the form `a = 1`, `a = 2`, ..., `a = 10_000`. Parsing this input yielded 30,000 AST nodes. Partial evaluation generated 409,994 lines of specialized C code, including comments. The resulting C code was compiled in 16.9 seconds on the x86_64 machine described in subsection 4.3. Handling the compilation time remains an open issue.

5 Discussion

5.1 Limitations

ASTro achieves good performance at low engineering cost, but the design choices that enable this simplicity also set boundaries on what the framework can achieve.

C as the Backend Ceiling. Unlike a hand-written JIT compiler that can emit arbitrary machine instructions, ASTro generates C code and therefore inherits the semantic and performance limitations of the C language and its toolchain. Low-level operations—such as deoptimization or instruction-set-specific optimizations—are beyond its reach.

Compiler-bound optimization quality. Since all heavy optimization is delegated to the C compiler, runtime speed cannot exceed the optimizer's capabilities. Differences in inlining heuristics, vectorization thresholds, or register allocators can translate directly into performance variance across platforms and compiler versions. In our experiments, we did not provide any parameters related to inlining, but for larger-scale programs, compiler-specific parameter tuning will likely be required.

Exception handling costs. Implementing exceptions in C essentially requires either (i) checking an error flag after every function call or (ii) using `setjmp`/`longjmp` for non-local exits. The former introduces a strict rule for users; the latter adds a `setjmp` frame to every protected call site, which can negatively impact performance or interfere with optimizations such as tail-call elimination. C++'s `try/catch` is one option.

Reflections on the Limitations. As discussed, ASTro faces performance limitations and cannot reach the peak speeds achieved by hand-crafted JIT compilers. However, its core advantage lies in its low adoption cost: users can employ ASTro with minimal effort—particularly C programmers, who require little additional system-specific knowledge—while still achieving respectable performance. Unlike JIT systems, which often require significant engineering effort, ASTro provides a lightweight and accessible approach to code generation. Furthermore, the generated C code is clear and human-readable, making debugging more straightforward than with native compilers when necessary.

5.2 Code-Cache Considerations

Code-size explosion. Partial evaluation is notorious for generating large amounts of specialized code when applied indiscriminately. ASTro has not yet addressed this issue, and we intend to explore well-known mitigation techniques in future work.

One practical approach to control inlining is to customize the hash function for specific node types. For example, a numeric literal node could return a unique hash for small constants (so they remain inlined), but map large constants to a fixed value like 0, encouraging reuse of a default dispatcher.

Context-aware hashing. The current cache key depends solely on AST structure. In practice, some runtime context will matter; the open question is *which* context to fold in. We plan to survey existing context-sensitive schemes before designing an ASTro-specific variant.

Hash safety. We use MurmurHash3[1] for speed, but have not yet analysed the consequences of hash collisions. A cryptographic hash would improve safety at the cost of extra CPU cycles—another trade-off that deserves empirical study.

Process-portable literals. Because ASTro embeds literal data directly into the specialized code, data whose address is process-specific (e.g. a pointer to the interned strings) cannot yet be shared across processes. We are investigating *COPY&PATCH*[16], which materialises process-specific data lazily after the code has been loaded.

Repository architecture. Modern cloud deployments often execute the same application across multiple processes or nodes. We envisage a dedicated *code repository* that answers "do you have a code named hash h?" queries. If the compiled fragment exists, the repository sends the code back;

[7] In the `prime_count` benchmark, the `gcc/-O2` build is unexpectedly slower than `gcc/-O1`, indicating a possible inlining pathology in GCC's optimizer.

otherwise the interpreter may send the AST and request compilation. Using Merkle hashes as keys lets the same protocol work consistently—from in-process caches, through cross-machine clusters, and out to geographically distributed datacenter deployments at almost astro-scale.

Cache Granularity. Determining *what* to store and *at which granularity* remains an open design question. For widely used libraries, it may suffice to cache entire function- or method-level subtrees. Caching every possible subtree, however, risks explosive growth; alternatively, one could focus on leaf-proximate subtrees that occur frequently. Building a real-world code repository and observing application workloads will reveal the optimal balance between cache coverage and storage cost.

5.3 Applicability

Static type systems remain unexplored. The present evaluation focuses on a dynamically-typed language; we have not yet experimented with a statically-typed one. Since ASTro makes no assumptions about types, a separate type-checking phase or type annotations on node definitions would need to be introduced.

In a statically-typed setting, each node would produce a typed result, but it is not immediately clear how well the current specializer–dispatcher scheme can accommodate this. Whether the framework can propagate static types to enable further optimizations, or whether it would require significant extensions (e.g., type-parameterized node variants), remains an open question for future work.

Backend language. In this work, we implemented ASTro using C as the backend language. Whether other languages can serve as viable backends remains an open question. In practice, the usefulness of a backend will largely depend on how aggressively its compiler performs inlining and related optimizations.

6 Related Work

This section reviews previous systems that intersect with ASTro.

RPython. PyPy's translation toolchain specializes an interpreter written in a restricted subset of Python [3, 12]. An abstract interpreter infers types and effects, after which multiple transformation passes emit a translated code and a meta-tracing JIT. ASTro adopts the same "single interpreter as ground truth" philosophy, but eliminates the complicated stages by off-loading low-level optimization to a commodity C compiler.

Futamura projections. The classic work of Futamura [5] formalized the notion that specializing an interpreter with respect to a source program yields a compiler. Subsequent surveys, most notably by Jones *et al.* [6], systematized offline vs. online partial evaluation and introduced binding-time analyses.

Truffle/Graal. The Truffle framework specializes an AST traversal interpreter at runtime and the Graal compiler generates high performance native code [13–15]. Profiling data guides partial evaluation within the JVM. ASTro likewise generates node-specific code, but delegates all machine-code generation to an external C compiler, thereby avoiding a JVM/Truffle/Graal dependency.

Truffle also supports advanced profiling and inline caches (e.g., Dispatch Chains [9]). It is still unclear whether ASTro can achieve such advanced optimizations with commodity C compilers.

AST vs. Bytecode. Larose *et al.* challenge the folklore that bytecode interpreters outperform AST-based ones by implementing both styles. They show that AST variants match or slightly exceed bytecode in raw and JIT-compiled speed, though bytecode remains more compact at large scale [7]. Our work confirms that AST-based runtimes deserve reconsideration within meta-compilation systems. Furthermore, our representation based on the Merkle-tree hash naturally enables the supernode optimization identified in that paper.

Cython. Projects like Cython [2] lower a subset of high-level languages to C (or `llvm-ir`) to piggy-back on mature toolchains. ASTro shares the same "let GCC/Clang do the heavy lifting" insight, but differs in that it emits *interpreter specializations* rather than transpiling.

Vmgen. Like Vmgen [4], which generates a complete efficient virtual machine in C from declarative descriptions of its byte-code instructions, ASTroGen produces an entire tree-traversal interpreter from declarative node specifications. In other words, Vmgen specializes on VM instructions, whereas ASTroGen specializes on AST nodes; both approaches let the author describe behavior once and obtain the supporting runtime infrastructure automatically.

Cross-process Code Caching. Mehta *et al.* demonstrate that JIT-compiled methods can be persisted in an off-line repository and reloaded in later executions when the dynamic context matches, cutting warm-up time [10]. ShareJIT extends the Android Runtime with a global code cache that allows JIT fragments to be shared across applications and processes by restricting the optimizer so that the resulting machine code is context-agnostic [17]. All of these systems key the cache by a mix of runtime context and compiler flags. ASTro applies the Merkle-tree idea at the granularity of *AST sub-trees*, enabling cross-process reuse of specialized code—something traditional build caches cannot exploit because they lack structural information about the AST.

7 Conclusion

Building a programming language interpreter from interpreter definitions alone is often caught between two extremes: hand-written interpreters, easy to modify but slow, and partial-evaluation frameworks, high-performance but complex, heavyweight, and platform-specific. **ASTro** bridges this gap by using a commodity C compiler as its backend. A key insight is the separation of dispatchers from per-node evaluators. This design allows the partial evaluator to generate specialized code simply by creating new dispatchers, without modifying the original user-defined node definitions. The resulting C code is structured to enable aggressive inlining by C compilers. By emitting plain, optimizable C code and naming each specialized AST sub-tree with a Merkle-tree hash for cross-process reuse, ASTro preserves the interpreter-only authoring model while achieving competitive performance.

Our naruby case study demonstrates that an evaluator and partial evaluator for a small dynamically-typed language can be implemented in just 300 lines of handwritten code. ASTroGen automatically expands this into approximately 2,500 lines of C source code. Ahead-of-time (AOT) compilation achieves performance close to that of `gcc/-O0`, while profile-guided (PG) compilation delivers further improvements—doubling performance over AOT on the `call` benchmark in the RISC-V environment.

Future Work. We plan to extend ASTro in three directions. First, supporting richer language features such as objects, exceptions, and garbage collection will help evaluate the framework's scalability, and allow us to assess how known optimizations can further improve performance. Second, we aim to explore JIT compilation within the ASTro framework. Finally, a shared, networked cache could allow large-scale deployments to amortize compilation costs across machines or edge devices.

Acknowledgments

We would like to thank Prof. Hidehiko Masuhara (Institute of Science Tokyo) for his valuable comments in the early stages of this research. We would also like to thank Benoit Daloze (Oracle Labs) for reviewing this paper and providing helpful comments, especially regarding Truffle/Graal.

A Toy language Declarations

ASTroGen is based on the following basic types. Additional fields may be added to store source location or profiling information.

```
// The user needs to provide VALUE definition.
typedef struct Node NODE;
typedef VALUE (*node_dispatcher_func_t)
                (CTX *c, NODE *n);
typedef uint64_t node_hash_t;
typedef node_hash_t (*node_hash_func_t)(NODE *n);
typedef void (*node_specializer_func_t)
                (FILE *f, NODE *n);
typedef void (*node_dumper_func_t)
                (FILE *f, NODE *n, bool online);

struct NodeKind {
    const char *default_dispatcher_name;
    node_dispatcher_func_t default_dispatcher;
    node_hash_func_t hash_func;
    node_specializer_func_t specializer;
    node_dumper_func_t dumper;
};

struct NodeHead {
    struct NodeFlags {
        bool has_hash_value;
        bool is_specialized;
        bool is_specializing;
          // to prohibit recursive specializing
        bool is_dumping;
          // to prohibit recursive dumping
    } flags;
    const struct NodeKind *kind;
    struct Node *parent;
    node_hash_t hash_value;
    const char *dispatcher_name;
    node_dispatcher_func_t dispatcher;
};
```

Given a definition file such as `node.def` (listed in Figure 2), ASTroGen generates the following C declarations.

```
struct node_num_struct {
    int32_t 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;
};
```

References

[1] Austin Appleby. 2011. MurmurHash3. https://github.com/aappleby/smhasher. Accessed: 2025-07-21.

[2] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engg.* 13, 2 (March 2011), 31–39. doi:10.1109/MCSE.2010.118

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/1565824.1565827

[4] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.* 32, 3 (March 2002), 265–294. doi:10.1002/spe.434

[5] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 381–391. doi:10.1023/A:1010095604496

[6] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., USA.

[7] Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. 2023. AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 233 (Oct. 2023), 29 pages. doi:10.1145/3622808

[8] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters. *SIGPLAN Not.* 50, 10 (Oct. 2015), 821–839. doi:10.1145/2858965.2814275

[9] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 545–554. doi:10.1145/2737924.2737963

[10] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (Oct. 2023), 22 pages. doi:10.1145/3622839

[11] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*. Springer-Verlag, Berlin, Heidelberg, 369–378.

[12] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 944–953. doi:10.1145/1176617.1176753

[13] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. doi:10.1145/2384716.2384723

[14] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.* 52, 6 (June 2017), 662–676. doi:10.1145/3140587.3062381

[15] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/2384577.2384587

[16] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 136 (Oct. 2021), 30 pages. doi:10.1145/3485513

[17] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT code cache sharing across processes and its practical implementation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 124 (Oct. 2018), 23 pages. doi:10.1145/3276494

Received 2025-07-20; accepted 2025-08-11