Automatically Generating an Abstract Interpretation-based Optimizer from a DSL

Ken Jin Ooi*
e0958256@u.nus.edu
National University of Singapore
Quansight
Singapore

Abstract

Just-in-Time (JIT) compilers can gain information at run time that are not available to Ahead-of-Time (AOT) compilers. As such, abstract interpretation baseline JIT compilers are common in many dynamic language implementations. Yet the reference implementation of Python — CPython, has largely avoided implementing a baseline JIT compiler, likely due to the prohibitive maintenance costs associated with one. This paper implements an abstract-interpretation based optimizer for CPython bytecode that is easy to maintain and less error-prone by automatically generating the optimizer from a pre-existing Domain Specific Language (DSL) — reusing the same DSL used to specify the interpreter. The key insight presented in this paper is that the very same DSL used to generate a concrete interpreter can also generate an abstract interpreter, providing multiple benefits such as being less error-prone and greater extensibility. The proposed abstract interpreter has been accepted into CPython 3.13 and forms a part of its experimental JIT compiler

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Domain specific languages**.

Keywords: Abstract Interpretation, JIT compiler, DSL

1 Introduction

Dynamic language implementations often employ JIT compilers to achieve performance competitive with statically typed language implementations [1]. Yet implementing JIT compilers for industrial-strength languages is no insignificant feat. Modern JIT compilers often require years of efforts and millions of lines of code. For example, V8, a highly optimized JavaScript runtime with a JIT compiler was implemented in over 2 million lines of code (counted with cloc [5]) over 16 years. This high implementation effort partly explains why even reference implementations of popular languages like Python lack JIT compilers. Handwritten optimizers are also susceptible to errors that arise from difference in the semantics understood by implementers.

To tackle the problem of high implementation effort of JIT compilers in the context of CPython, we propose reusing the DSL that CPython already implements to specify its bytecode definitions, to automatically generate an abstract interpretation-based [4] optimizer — an integral part of many JIT compilers.

Abstract interpretation-based optimizers are a key component of baseline JIT compilers. They are found in many baseline JIT compilers today for WebAssembly [11]. These optimizers often perform simple optimizations that can be done without significant computational cost. Their main idea is that by abstracting program information, we can gain information about a program even without executing the program. In the case of CPython, before our efforts, no such JIT compiler optimizer existed.

While our implementation is specific to CPython, it is possible to extend this technique to all baseline JIT compilers with an interpreter definition. Additionally, this paper takes only the first (but important!) step towards a competitive runtime. Competitive runtimes like V8 often employ multiple tiers of JIT compilation, with increasingly sophisticated optimizations progressively for each tier. We thus do not expect performance to be competitive with PyPy [2] or GraalPy (based on the Truffle [12] framework).

To evaluate the effectiveness of our optimizer, we measure performance uplifts. Preliminary results suggest up to a 6% speedup in certain benchmarks.

2 Brief Background of CPython 3.13

CPython 3.13 includes an experimental runtime JIT compiler. A DSL based on vmgen [6] and HotPy [9] is used to specify the interpreter. This DSL specifies in its header its input stack effect, output stack effect, and its body consists of the actual C instructions to execute. For example, the following is the DSL specification for the instruction `LOAD_FAST` (simplified from the original in CPython for better illustration). `LOAD_FAST` loads a local variable `value` onto the operand stack.

```
1 inst(LOAD_FAST, (-- value)) {
2     value = GETLOCAL(oparg);
3     assert(value != NULL);
4     Py_INCREF(value);
5 }
```

In this work, we propose optimizing CPython's implementation through an abstract interpreter, the core insight being

*Other contributors have contributed additional optimizations on top of the initial version, as CPython is an open-source project.

that the DSL can be extended and re-used to generate this interpreter automatically.

3 Approach

The key insight behind our approach is that we can utilize the current specification of the bytecode semantics in the DSL to automatically generate an optimizer, tackling the aforementioned error-proness and maintenance burden. The approach distinguishes between two main cases, sound default rules, and optimized custom rules.

Sound default rules. The main requirement of an abstract interpreter is soundness (that is, the abstract domain accurately represents the concrete domain, and the information gain is monotonic). The default abstract interpreter generated for CPython 3.13 is sound but gains no information. This paper generates these cases by simply reading the operand stack effect from the instruction's DSL, then setting modified stack values as either `not-null` or `unknown`. The vast majority of abstract interpreter cases are automatically generated.

The default C code generated for the abstract interpreter for our `LOAD_FAST` instruction would be the following. Note that this is generated from the stack effect `-- value` which indicates that `value` is pushed to the operand stack:

```
case _LOAD_FAST: {
    _Py_UopsSymbol *value;
    value = sym_new_not_null(ctx);
    stack_pointer[0] = value;
    stack_pointer += 1;
    break;
}
```

Assuming the original DSL for `LOAD_FAST` is sound, the generated abstract rule for `LOAD_FAST` will also be sound.

Optimized custom rules. To apply custom optimizations, the abstract interpreter allows for overriding the original interpreter's DSL with its own custom DSL. This allows for optimizations like constant propagation and guard removal. This enables the abstract interpreter to be easily extensible, and for gradual migration from the default bytecode semantics to custom optimized ones. The developer does not need to care about the entire abstract interpreter or implement the entire interpreter all in one go initially. Instead, they are able to gradually add more custom DSL rules to the abstract interpreter as more optimizations are targeted.

4 Related Work

SpecTec [3] is a DSL for specifying small-step semantics in WebAssembly. Its goals differ from ours as it generates WebAssembly specification artefacts, while we generate an optimizer.

This work was also directly influenced by HotPy's Glasgow Virtual Machine Toolkit [9] project which used a similar DSL for optimizations. The key difference is its scope. HotPy is a complete re-implementation of Python, while our project incrementally improves CPython. Our project works within the constraints of the existing CPython infrastructure and re-uses its prior base interpreter definition, while HotPy is allowed to freely implement whatever interpreter definition it so chooses. Implementing an optimizer on top of a 30-year-old codebase for an industrial-strength virtual machine also poses significant engineering and backward compatibility challenges not present in a ground-up implementation.

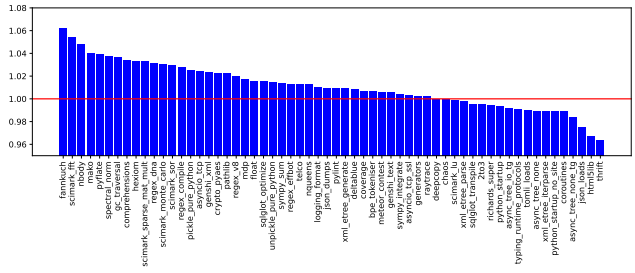

Figure 1. Optimizer on versus off. Speedups as a fraction (Non-significant results hidden). Results over 1.00 indicate a speedup, results below 1.00 indicate slowdown.

5 Evaluation

We evaluate the optimizer in the aspects of implementation effort and performance. For implementation effort, the generator for the abstract interpreter is only 196 lines of Python code (measured via `cloc`), while the DSL and C code for the abstract interpreter is around 1100 lines of code . To measure performance, we use `pyperformance` [10] — the benchmark suite used by CPython, with more benchmarks added in, and some microbenchmarks removed. `pyperformance` automatically runs benchmarks multiple times to account for fluctuations. We conduct a Mann-Whitney U-test [7] to measure significance of the results. Overall, a geometric mean 1% speedup was reported by `pyperformance` using Ubuntu 20.04 and `gcc` 9.4.0, with speedups ranging up to 6% (Fig 1) (artefacts available [8]). This is significant considering the optimizer is still work-in-progress and most of the optimizations are not yet implemented.

6 Conclusion

In summary, we tackle the problem of unwieldy optimizers by generating an abstract interpreter automatically from a DSL. This work has already been adopted by CPython 3.13 [1] and is part of its experimental JIT compiler. More work is planned to extend the abstract interpreter's capabilities and optimizations performed. This work can also extend to a register machine. Formally verifying and specifying the abstract interpreter would be an interesting endeavour.

[1] https://github.com/python/cpython/pull/115085

References

[1] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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*, pages 18–25, 2009.

[3] Joachim Breitner, Philippa Gardner, Jaehyun Lee, Sam Lindley, Matija Pretnar, Xiaojia Rao, Andreas Rossberg, Sukyoung Ryu, Wonho Shin, Conrad Watt, and Dongjun Youn. Wasm spectec: Engineering a formal language standard, 2023.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[5] Albert Danial. cloc, 2024.

[6] M Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen—a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 32(3):265–294, 2002.

[7] HB Mann and DR Whithney. On a test of whether one of two random variables is stochastically larger than the other ', annuals of mathematical statistics, 18. 1947.

[8] Ken Jin Ooi. Automatically generating an abstract interpretation-based optimizer from a dsl artefacts, 2024.

[9] Mark Shannon. *The construction of high-performance virtual machines for dynamic languages*. PhD thesis, University of Glasgow, 2011.

[10] Victor Stinner. pyperformance, 2024.

[11] Ben L. Titzer. Whose baseline compiler is it anyway?, 2023.

[12] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.