Seeing Through the FFI: Analyzing Native Code in Dynamic Languages

Filip Říha
rihafili@fit.cvut.cz
Czech Technical University, Prague, Czechia

Abstract

Dynamic languages such as Python, Ruby, or R rely heavily on native C/C++ functions using a foreign function interface. While they significantly speed up baseline performance, these functions are opaque to a Just-in-time compiler — parameter and return types, side effects, and argument counts are not exposed for optimization. We propose an approach that analyzes native code in the context of the host VM, opening up the semantics of native code without the need for manual transformation. From this we identify three use cases: a) inferring precise types and side effects for JIT specialization, b) modeling the dynamic language semantics in the same IR, enabling whole-program analysis, and c) extracting specialized native code slices that only capture the main logic for subsets of inputs.

ACM Reference Format:
Filip Říha. 2026. Seeing Through the FFI: Analyzing Native Code in Dynamic Languages. In . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivation

High-level programming languages like Python, Ruby, or R usually depend heavily on large collections of functions implemented in system-level languages like C, C++, or Fortran. This reliance comes from the continuous discrepancy in performance between the high-level languages and the system-level compiled ones. To mitigate this gap, language implementers and library authors rely on the *foreign-function interface* (FFI), which allows performance-critical functionality to be implemented natively and invoked from the dynamic language runtime.

Although FFI functions are needed to reach reasonable interpreter performance, for the virtual machine (VM) the code behind the native interface boundary becomes effectively a black-box. This arises from the differences between the levels of abstraction with which the languages work. While the high-level language works in the managed environment provided by the VM, the native functions work directly inside the context of the VM code. This means that it can arbitrarily change the state of the VM, including, but not limited to, changing the state of the call stack, allocating or destroying objects on the heap, or mutating existing values and variables. Since a value in a dynamic language can be of any type (often called a *boxed value*), the callee needs to first validate the shape of the arguments, before extracting the underlying representation (*unboxing* it) and using it. Even the number of arguments might be hidden from the runtime, since typically the signature of the native function takes an arbitrarily sized array of arguments, expecting the callee to raise a runtime error in case of mismatch. All of this behavior is hidden behind a very opaque interface — the VM only sees the symbol name being called. It has no idea about any side effects or the parameters and result types.

Let us take for example the call of plus on two values (`x + y`) in the context of the R programming language. The semantics of addition in R are quite complicated. It is defined on integers, floats, and any combination of them, possibly performing a type coercion. The number types in R are vectorized, performing the operation elementwise, even for vectors of different sizes. Additionally, the operation may lead to an object-oriented dispatch, landing in arbitrary code. All of these behaviors lead to an implementation full of guards, checks and unboxings. To further complicate the logic, heuristic-based "fast paths" are often part of the built-ins. In the case of addition in R, it first checks if both values are simple scalar integers, and if true, performs a fast C integer addition.

All of this culminates at the point of *Just-in-time* (JIT) compilation. When compiling a call to a native function, we need to preserve all of its semantics. By simply emitting a call to the native function, the JIT does not get much speedup, since it still needs to go through all the checks and branches inside the function. Even after inlining the callee and using generic intermediate representation (IR) optimizations (*e.g.* the LLVM or GCC infrastructure), the gains are insignificant — without domain-specific information, these optimizations will for example not elide the creation of temporary objects, as from the native code convention they are technically observable side effects. Only by leveraging the extra inferred information, gained either by static analysis or *speculative assumptions* based on observed feedback, can significant speedup be made. Values can be unboxed, type checks can be omitted, and constants can be folded. But the problem still remains: *How to leverage the information about the native function in the compilation context?*

The current approach is very manual. By observing the implementation of the function, one can derive the input and output types and side effects of the function. To optimize for some input types, a specialized version needs to be manually rewritten (usually in the IR), duplicating the semantics of the function. This can be not only error-prone, but also very tedious — for example, the R virtual machine contains almost 800 built-in functions. Furthermore, this manual work is typically focused on the functions provided by the virtual machine, completely disregarding the native code provided by libraries and packages.

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.
Conference'17, Washington, DC, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Breaking the compilation barrier. The main idea of this proposal is to break the barrier between untyped highly dynamic languages and their native code parts. We want to develop a framework to automatically analyze functions written in system-level languages and utilize the insights gained by understanding the code in the context of the VM to improve both runtime performance and correctness.

2 Proposed Approach

The research will focus on the R programming language and its native functions, both built into the VM and provided through packages and libraries. The first step is to create and use an intermediate representation capturing the high-level semantics of both C/C++ and the R virtual machine. Using this IR, we can further analyze and optimize calls to the native code. Currently, we propose these usages:

Inferring types and effects. By analyzing the native code, we will be able to infer correct and precise types, which can be further leveraged by a JIT compiler. Without any information, any call to C-level function can arbitrarily change the state of the VM runtime, invalidating all assumptions after that point. By having a more precise function types that also capture side effects, the optimizations can be more effective by possibly keeping previous assumptions, while also inferring a more precise result type.

Unified intermediate representation. Creating an IR that is capable of capturing all the necessary semantics of not only native functions but also the R language, we would be able to analyze and optimize the whole program, allowing for truly whole program analysis across language boundaries. This would also entail using domain-specific optimizations to further speed up the runtime without sacrificing correctness or manual labor.

Extracting specialized native code slices. Alternatively, given the inferred range of valid inputs to a native function, multiple specialized *code slices* can be generated. This can be done by fixing the inputs to some subset of its and running optimizations, eliminating redundant type checks and guards in the process. The final slice would only contain the actual logic of the function. Such slices could be then leveraged not only by a JIT compiler, but also by an optimizing interpreter. As an example, the R function `do_arith` responsible for executing all unary and binary operations, which spans more than 300 lines mostly consisting of type guards and error reporting, could be simplified into the following code[1]:

```
// SEXP is the boxed type of R values
SEXP specialized_add(SEXP x, SEXP y) {
  SEXP ans = ALLOC_VECTOR(INTSXP, 1);
  SET_SCALAR_IVAL(ans, SCALAR_IVAL(x) +
      SCALAR_IVAL(y));
  return ans;
}
```

Or, for the case that we already unboxed the values, an even simpler slice can be generated:

```
int specialized_add(int x, int y) {return x + y;}
```

[1]Note that for the sake of brevity an overflow check is omitted

The challenge of this approach lies in identifying the interesting argument type subsets, as well as preventing combinatorial explosion of generated code slices. It is also unclear what format the generated slices should take, as they would need to be audited, tested, and possibly generated as part of the build step.

The current plan is to use the *MLIR* [1] framework for the analysis, and the Clang compiler and its MLIR-based ClangIR dialect as a base for the new IR.

3 Related Work

The *MJIT* [2] compiler for Ruby compiles both the interpreter and C extension code into a single IR called MIR [3], enabling inlining and optimization across the FFI boundary. However, only generic optimizations are applied. No domain-specific knowledge about the host language's type system or runtime semantics is exploited.

GraalVM [4] with the Truffle framework enables polyglot programming by hosting multiple language runtimes in a single VM. Its LLVM bitcode interpreter Sulong allows native C code to participate in Graal's speculative JIT optimizations. This requires native code to be available as LLVM bitcode at runtime, and optimizations are still driven by general transformations rather than by knowledge of the dynamic language's semantics.

Different technique used in PyPy [5], an alternative Python runtime, is called *typed methods* [6]. Manual annotations on C extension functions expose their parameter and return types to the VM, while the type checking and unboxing is done by the runtime. This allows the JIT compiler and even the interpreter to eliminate boxing and unboxing overhead on C extension calls where possible. However, this approach is entirely manual and requires modification to the original source code.

Acknowledgment. The research is supported by Czech Science Foundation Grant No. 23-07580X.

References

[1] Chris Lattner et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2–14. doi:10.1109/CGO51591.2021.9370308.

[2] Vladimir Makarov. 2023. An MIR-based JIT prototype for Ruby. https://developers.redhat.com/articles/2023/10/09/mir-based-jit-prototype-ruby.

[3] Vladimir Makarov. 2020. MIR: A lightweight JIT compiler project. https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project#.

[4] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2013). Association for Computing Machinery, Indianapolis, Indiana, USA, 187–204. ISBN: 9781450324724. doi:10.1145/2509578.2509581.

[5] 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* (ICOOOLPS '09). Association for Computing Machinery, Genova, Italy, 18–25. ISBN: 9781605585413. doi:10.1145/1565824.1565827.

[6] Maxwell Bernstein and Carl Friedrich Bolz-Tereick. 2024. Dr Wenowdis: Specializing Dynamic Language C Extensions using Type Information. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (SOAP 2024). Association for Computing Machinery, Copenhagen, Denmark, 1–8. ISBN: 9798400706219. doi:10.1145/3652588.3663316.

Czech Technical University
Thákurova 9
160 00 Praha 6
Czech Republic



Prague, 15.4.2026

Acknowledgement

To whom it may concern,

Hereby I confirm that I am aware of the fact that Filip Říha, first year PhD student that I advise, is submitting a research proposal to the 2026 doctoral symposium collocated with ECOOP.



Filip Křikava
Associate Professor
Programming Language Laboratory
Faculty of Information Technology