Seeing Through the FFI: Analyzing Native Code in Dynamic Languages

Filip Říha

Czech Technical University

Motivation

An example from the R land

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, 1, 1.5, 2.5, 3.5, 5.5, 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```

- Store vector of numbers
- Multiply all numbers by 2
- Sum all numbers

Motivation

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, 1, 1.5, 2.5, 3.5, 5.5, 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```

After executing several times, we would like to JIT compile
To reasonably optimize, we want to inline `c`, `vapply`, and `sum`:

Motivation

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, 1, 1.5, 2.5, 3.5, 5.5, 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```

After executing several times, we would like to JIT compile
To reasonably optimize, we want to inline `c`, `vapply`, and `sum`:

```
c <- function (...)      .Primitive("c")
vapply <- function (...) .Internal(vapply(...))
sum <- function (...)    .Primitive("sum")
```

Motivation

```
c <- function (...)       .Primitive("c")
vapply <- function (...) .Internal(vapply(...))
sum <- function (...)     .Primitive("sum")
```

`.Primitive` and `.Internal` call into *native (C) code*

Motivation

```
c <- function (...)       .Primitive("c")
vapply <- function (...) .Internal(vapply(...))
sum <- function (...)     .Primitive("sum")
```

`.Primitive` and `.Internal` call into *native (C) code*
Actual signature of functions is

```
SEXP do_c       (SEXP call, SEXP op, SEXP args, SEXP env);
SEXP do_vapply  (SEXP call, SEXP op, SEXP args, SEXP env);
SEXP do_summary (SEXP call, SEXP op, SEXP args, SEXP env);
```

(*SEXP is an R boxed object*)

This does not tell us anything about *input types*, *output types* or *effects* (argument mutation, reflection, IO, ...)

Motivation

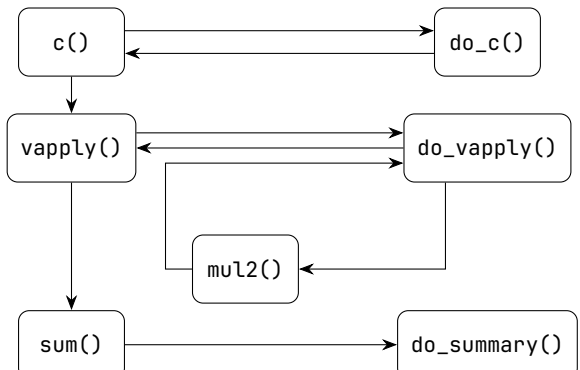
Native functions have access to the whole VM and can change the world in unpredictable ways

```
x <- 42
.Call("change_world") # Native function call
print(x)
```

There is no guarantee this will print 42

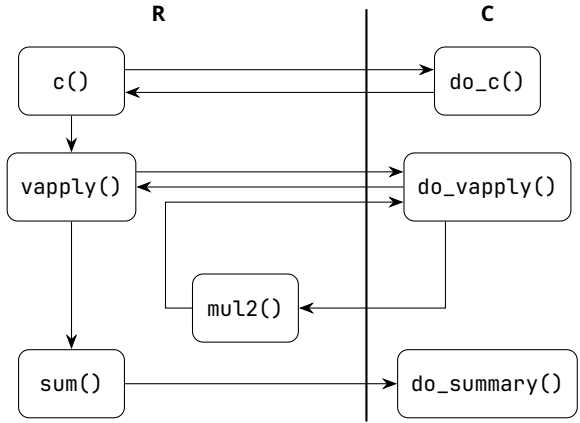
Motivation

```r
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, ..., 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```



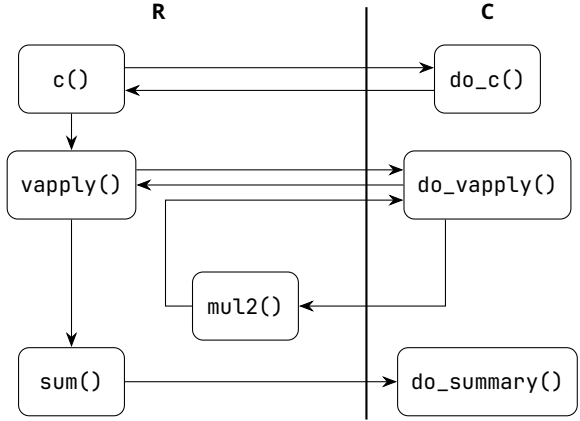
Motivation

```r
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, ..., 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```



Motivation

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, ..., 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```



We have an *optimization barrier* between the languages

A universal problem in VMs

Ruby

C functions dominate execution time

Only 10% of execution time is spent in JIT code

Aggregated Event Data:

[dso]	[symbol or category]	[top-most cycle ratio]	[num cycl...
ruby	[interpreter]	16.83%	4716462794
sqlite3_native.so	[sqlite3]	15.77%	4418942723
Multiple DSOs	[GC]	15.60%	4371176801
perf-123171.map	[JIT code]	10.21%	2862379495
Multiple DSOs	[unknown]	3.18%	890849102
ruby	[regexp]	2.62%	733409123
Multiple DSOs	[sha256]	2.43%	681741273
ruby	[rb_str_*]	2.24%	627899901
ruby	[rb_hash_*]	2.12%	593106336
Multiple DSOs	[ivars]	1.99%	556585915
libc.so.6	__memmove_avx_unaligned_erms	1.60%	447907631
ruby	rb_id_table_lookup	1.59%	446611117
ruby	[rb_ary_*]	1.34%	376892555
ruby	[rb_st_*]	1.33%	374047549
ruby	find_table_entry_ind	1.27%	356375435
libc.so.6	[pthread lock]	0.91%	255626481

https://gist.github.com/k0kubun/5e0b3bb894e9fed9b01e25fd25e8bea5

Takashi Kokubun, *ZJIT: Building a New JIT Compiler for Ruby*, REBASE 2025

Python

Cross Module Quickening – The Curious Case of C Extensions

Felix Berlakovich
University of the Bundeswehr Munich, Neubiberg, Germany

Stefan Brunthaler
University of the Bundeswehr Munich, Neubiberg, Germany

Abstract

Dynamic programming languages such as Python offer expressive power and programmer productivity at the expense of performance. Although the topic of optimizing Python has received considerable attention over the years, a key obstacle remains elusive: C extensions. Time and again, optimized run-time environments, such as JIT compilers and optimizing interpreters, fall short of optimizing *across* C extensions, as they cannot reason about the native code hiding underneath.

To bridge this gap, we present an analysis of C extensions for Python. The analysis data indicates that C extensions come in different varieties. One such variety is to merely speed up a single thing, such as reading a file and processing it directly in C. Another variety offers broad access through an API, resulting in a domain-specific language realized by function calls.

While the former variety of C extensions offer little optimization potential for optimizing run-times, we find that the latter variety *does* offer considerable optimization potential. This optimization potential rests on *dynamic locality* that C extensions cannot readily tap. We introduce a new, interpreter-based optimization leveraging this untapped optimization potential called Cross-Module Quickening. The key idea is that C extensions can use an optimization interface to register highly-optimized operations on C extension-specific datatypes. A quickening interpreter uses these information to continuously specialize programs with C extensions.

To quantify the attainable performance potential of going beyond C extensions, we demonstrate a concrete instantiation of Cross-Module Quickening for the `CPython` interpreter and the popular `NumPy` C extension. We evaluate our implementation with the `MPBench` benchmark suite and report performance improvements by a factor of up to 2.84.

Possible solutions

Rewriting the builtins

- Approximates the semantics in a different language
- Unusable for libraries
- Typically used for a handful of specialized builtins

Possible solutions

Rewriting the builtins

- Approximates the semantics in a different language
- Unusable for libraries
- Typically used for a handful of specialized builtins

Annotating the function

- More load on library makers
- Might not be sound
- Easy to desync from the implementation

Possible solutions

Rewriting the builtins

- Approximates the semantics in a different language
- Unusable for libraries
- Typically used for a handful of specialized builtins

Annotating the function

- More load on library makers
- Might not be sound
- Easy to desync from the implementation

Unifying the execution runtimes

- Large engineering effort
- The unification usually loses large amount of detail
- Interpreting native code is hard and slow
- *e.g. Sulong*

Possible solutions

Rewriting the builtins

- Approximates the semantics in a different language
- Unusable for libraries
- Typically used for a handful of specialized builtins

Annotating the function

- More load on library makers
- Might not be sound
- Easy to desync from the implementation

Unifying the execution runtimes

- Large engineering effort
- The unification usually loses large amount of detail
- Interpreting native code is hard and slow
- *e.g. Sulong*

All of these require large and continuous engineering effort

Proposed solution

Framework for automated analysis of native functions

Proposed solution

Framework for automated analysis of native functions
in the context of the VM

Proposed solution

Framework for automated analysis of native functions
in the context of the VM

Connecting the *fundamentally mismatched* languages without losing important details about the semantics

`do_arith` function

the +, -, *, /, ... handler, ~800 LoC



```
SEXP do_arith(SEXP call, SEXP op, SEXP args, SEXP env) {
  SEXP arg1 = CAR(args), arg2 = CADR(args);

  if (IS_SCALAR(arg1, REALSXP) && IS_SCALAR(arg2, REALSXP)) {
    double x1 = SCALAR_DVAL(arg1);
    double x2 = SCALAR_DVAL(arg2);
    SEXP ans = allocVector(REALSXP, 1);

    switch (PRIMVAL(op)) {
      case PLUSOP: SET_SCALAR_DVAL(ans, x1 + x2); return ans;
      // ...
    }
  }
  // ...
}
```

simplified snippet - fast path for scalar double addition

Issues — Types

```
typedef struct SEXPREC *SEXP;
```

There is a fundamental *mismatch* between C and R types
C stores the type as *runtime data*, but some sort of *compile time information* is required

Issues — VM Manipulation

```
SEXP ans = allocVector(REALSXP, 1);
```

We need to model R VM operations, as C has no idea about R heap, objects, values or effects

Issues — Effects

```
SEXP do_arith(SEXP call, SEXP op, SEXP args, SEXP env) {
  if (argc != 2 && argc != 1)
    errorcall(call,_("operator needs one or two arguments"));
  // ...
}
```

Can the function fail?
Does it do IO?
Does it modify the R runtime (global state, reflection, ...)?

Goals — Modeling the semantics

The first step is to model the VM semantics at a reasonable level (operations, types, effects)

```
allocVector(REASXP, 1);   // pure, return dbl1

SET_SCALAR_DVAL(sxp, 42); // pure, (dbl1, c_dbl) -> void

errorcall(call,_("operator needs one or two arguments")); // errors
```

Goals — Type and effect inference

```
SEXP do_arith(SEXP call, SEXP op, SEXP args, SEXP env) {
  // ...
  if (IS_SCALAR(arg1, REALSXP) && IS_SCALAR(arg2, REALSXP)) {
    // ...
    SET_SCALAR_DVAL(ans, x1 + x2);
    return ans;
  }
  // ...
}
```

For this fast path we can infer it is *pure* and its type is `(dbl1, dbl1)` → `dbl1`

Inferring precise, yet useful types and effects is the most crucial step for this research

Goals — Bug finding

Domain-specific analysis can be used to find bugs in low-level code

```
SEXP x = allocVector(REALSXP, 1); // it is not rooted
SEXP y = allocVector(REALSXP, 1); // can trigger GC
// x could be collected here
send(x, y);
```

Could replace existing *RCHK*[1] LLVM-based tool

[1] https://github.com/kalibera/rchk

Goals — Specialized native code slices

For a specific call

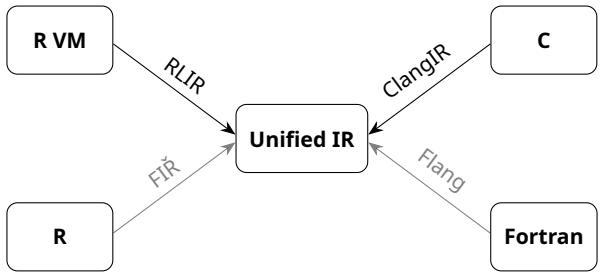
```
3.0 + 4.0
```

we could dispatch to an optimized function slice

```
SEXP do_arith_v1(SEXP arg1, SEXP arg2) {
    double x1 = SCALAR_DVAL(arg1);
    double x2 = SCALAR_DVAL(arg2);
    SEXP ans = allocVector(REALSXP, 1);
    SET_SCALAR_DVAL(ans, x1 + x2);
    return ans;
}
```

Goals — Unified IR

The whole IR ecosystem could be extended by more dialects, allowing truly whole-program analysis



MLIR



- *Multi-Level Intermediate Representation*
- IR is composed of individual *dialects*, which can be mixed and composed
- First-class support for well-formedness, transformations, analysis and optimizations

```
module {
  func.func @main() -> i32 {
    %0 = arith.constant 42 : i32
    return %0 : i32
  }
}
```

https://mlir.llvm.org/

ClangIR

- C/C++ MLIR dialect
- Part of the *Clang* compiler
- Models the languages at a high level

```
SEXP do_arith(/*...*/) {
  SEXP arg1 = CAR(args),
       arg2 = CADR(args);

  if (IS_SCALAR(arg1, REALSXP)
    && IS_SCALAR(arg2, REALSXP)) {
    double x1 = SCALAR_DVAL(arg1);
    // ...
  }
}
```

```
!SEXP = !cir.ptr<!cir.record<struct "SEXPREC">>
cir.func @do_arith(...) -> !SEXP {
  %0 = cir.call @CAR(%args)
  %1 = cir.call @CADR(%args)
  %3 = cir.const #cir.int<14> // REALSXP
  %4 = cir.call @IS_SCALAR(%0, %3)

  %5 = cir.ternary(%4, true {
    %6 = cir.call @IS_SCALAR(%1, %3)
    cir.yield %6
  }, false {
    %6 = cir.const #false
    cir.yield %6
  }) // &&

  cir.if %5 {
    %6 = cir.call @SCALAR_DVAL(%0)
  }
}
```

https://llvm.github.io/clangir/

RLIR[2] *(in progress)*

R Level IR

New MLIR dialect modeling the R types, effects, and operations

```
cir.call @CAR(%args)
  : (!SEXP) -> !SEXP

%3 = cir.const #cir.int<14>
cir.call @IS_SCALAR(%0, %3)
  : (!SEXP, !i32) -> !cir.bool

cir.call @SCALAR_DVAL(%0)
  : (!SEXP, !i32) -> !cir.double

%8 = cir.const #cir.int<1>
cir.call @allocVector(%3, %8)
  : (!i32, !i32) -> !SEXP
```

```
rlir.list.car(%args)
  : !rlir.sexp<LIST> -> !rlir.sexp<ANY>


rlir.is_scalar REAL (%0)
  : !rlir.sexp<ANY> -> !cir.bool

rlir.scalar_value REAL (%0)
  : !rlir.sexp<REAL,Scalar> ->
                              !cir.double

rlir.allocVector(REAL, length=1)
  : !rlir.sexp<REAL,Scalar>
```

[2] RIR was taken

And finally, we could get to a point where the initial example

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, 1, 1.5, 2.5, 3.5, 5.5, 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```

And finally, we could get to a point where the initial example

```
mul2 <- function(x) x * 2

example <- function() {
  xs <- c(0.5, 1, 1.5, 2.5, 3.5, 5.5, 6.5)
  ys <- vapply(xs, mul2)
  sum(ys)
}
```

could be optimized into

```
example <- function() 42
```