Kent Academic Repository

Full text document (pdf)

Citation for published version

Bertholon, Guillaume and Kell, Stephen (2019) Towards Seamless Interfacing between Dynamic Languages and Native Code. In: VMIL 2019, Athens, Greece. (In press)

DOI

Link to record in KAR

https://kar.kent.ac.uk/76576/

Document Version

Publisher pdf

Towards Seamless Interfacing between Dynamic Languages and Native Code

Guillaume Bertholon[*]
École Normale Supérieure
Paris, France
guillaume.bertholon@ens.fr

Stephen Kell
University of Kent
Canterbury, United Kingdom
S.R.Kell@kent.ac.uk

Abstract

Existing approaches to interfacing high- and low-level code push considerable burdens onto the programmer, such as wrapper maintenance, explicit code generation, interface re-declaration, and/or signalling to garbage collectors. We note that run-time information on data layout and allocations in native code is available, and may be extended with knowledge of object lifetimes to assist in automating garbage collection. We describe work in progress towards an extension of the CPython virtual machine along these lines. We report initial experience building a first working prototype, and some early performance experiments.

CCS Concepts • **Software and its engineering → Interoperability**; *Virtual machines*; *Garbage collection*; Allocation / deallocation strategies.

Keywords Python, FFI, garbage collection, debugging

ACM Reference Format:
Guillaume Bertholon and Stephen Kell. 2019. Towards Seamless Interfacing between Dynamic Languages and Native Code. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361230

1 Introduction

Programming is fragmented by language boundaries. High-level languages, especially those offering automatic garbage collection, have traditionally been implemented so as to interoperate with external code only via a 'foreign function interface' (FFI) designed to maximise implementer freedom rather than to help programmers. As has been argued before, such programmer-facing boundaries ideally ought not to exist at all [11] and are more cultural norm than technical necessity [6]. However, viable alternative implementation approaches remain elusive.

Both older and more recent work towards the elimination of FFIs has focused on 'polyglot VMs' [8, 13, 19], in which a shared core of compilation and run-time infrastructure hosts many languages. These are capable of impressive results, including high performance in mixed-language code. However, realising multiple languages atop a single core implementation has inherent scalability limits: no single such system will conquer all, and innovation demands freedom to rethink the core. It also brings difficulties with backwards compatibility, on both high- and low-level sides [9]. In order to fully embrace native code, polyglot VMs must reimplement or integrate not only support for languages such as C or C++ (or LLVM), but also large parts of the assemble-link-load toolchain [16].

Other efforts have accepted the presence of FFIs but focused on mitigating their difficulties, for example by generating FFI code [1], by automated processing of C header files [4], and/or by hosting glue logic within the higher-level language [12]. These systems are often successful in lowering the burden, but cannot eliminate it: maintenance difficulties are inherent (complex per-codebase 'binding code' still exists, and must be adapted as either FFIs or external APIs change), while the 'feel' of the host language is not preserved.

Relative to these two previous approaches, this paper explores a 'third way' distinguished by:

1. largely retaining existing language implementations rather than replacing them (hence allowing a high standard of compatibility, at source and ABI levels);
2. notwithstanding point 1, *extending the native-code toolchain* and its Unix-like run-time services, in order to support better the needs of high-level languages, rather than treating those services as a black box;
3. notwithstanding point 1, lightly modifying language VMs' implementations, particularly in order to retrofit them onto the extended runtime of point 2.

Our specific contributions are embodied in a CPython virtual machine extended towards seamless interoperability with native (currently C) code, roughly as follows.

[*] Work carried out during an internship at the University of Kent.

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361230

- We describe the basic design and construction of an 'FFI-less' extension to CPython which builds on liballocs [10]. This provides run-time type information (extracted via postprocessing of compiler-generated native debugging information) and also abstractions of *allocator* and *allocation* which provide useful building blocks (§3).
- We describe some idiomatic default 'language mapping' design choices exploiting the available run-time type information, and show how these balance a trade-off between natural usage in simple cases and 'least surprise' in more complex ones (§4).
- We present an approach to mediating reference-counted memory management with manual use of malloc() and free(). Unlike previous approaches, we explore the addition of a pointer write barrier into native code, and describe how this allows process-wide reference counting (§5). We also briefly describe how this extends to custom memory allocators or wrappers thereof (common in C code) and to the idiomatic treatment of C APIs that involve initialization or finalization.

2 Existing Approaches

Manual FFI coding Some libraries (such as Numpy[1]) are written as 'VM extensions' coded against the FFI of CPython. This allows flexibility, including adding native builtin functions or types to the language, but at the cost of manually wrapping all C-language functions and types exposed by the library interface (as seen in Listing 1).

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    int sts = system(command);
    return PyLong_FromLong(sts);
}
```

Listing 1. Exposing system() in a CPython extension

Generated FFI code To reduce this burden, some tools such as cffi behave as code generators for native function wrappers. However, the experience imports a lot of the complexity of programming in C, such as include files and ahead-of-time code generation. Most significantly, it remains necessary to *re-declare* the interface functions' arguments and return types via the code generator's API (as seen in Listing 2).

```
from cffi import FFI
ffibuilder = FFI()
ffibuilder.cdef("""float pi_approx(int n);""")
ffibuilder.set_source("_pi_cffi", """
    #include "pi.h"   // the C header of the
        library""",
    libraries=['piapprox'])   # library name,
        for the linker
if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

Listing 2. Example of generating wrappers for a C pi_approx() function using cffi

Data model embedding A similar but more 'on-line' approach is exemplified by ctypes, which embeds the C data model into Python and hides the step of code generation—instead, the system dynamically loads the referenced shared library on request by client Python code. However, the result is similar: the programmer must annotate or rather (re-)declare any required composite types (like structs, enums, and unions), as seen in Listing 3. Overall, C-style programming is still imported into Python.

```
from ctypes import *
libc = CDLL("libc.so.6")
libc.strchr.restype = c_char_p
libc.strchr.argtypes = [c_char_p, c_char]
libc.strchr(b'abcde', b'c') # => returns b'cde'
```

Listing 3. Example of a call to libc's strchr using ctypes

3 Outline Approach

None of the previous approaches offer a 'Pythonic' experience. For example, client code is concerned with passing variables by copy or by reference, and the ctypes example clearly shows how the C concept of pointers has become part of the Python-side data model. We would instead like C libraries to be seen in Python as regular Python modules, out of the box, needing little or no wrapper code or annotation.

3.1 Insights

We observe that several useful building blocks exist.

Dynamism in C Dynamism is available in C, albeit in system-defined interfaces rather than within the language. Dynamic loading [7] is provided by POSIX's dlopen() family of calls. On-line debugging is possible via metadata formats such as DWARF [5], providing rich descriptions of code, variables and types; an extended VM may consume these directly, without user-supplied wrappers or parsing C headers.

Reified type information liballocs [10] postprocesses DWARF into run-time type information on native libraries. Like DWARF, it is descriptive: it captures many realisations of recurring abstractions (such as data types and allocated memory objects), whereas a language virtual machine defines a single such realisation. This is why native debuggers need not share code with the compiler, unlike debug servers in virtual machines. Appendix B gives a condensed overview of liballocs.

[1] A very popular numerical library for Python: https://www.numpy.org/.

Run-time allocator protocol liballocs models memory at run time as an *allocation hierarchy*. Arbitrary pointers may be queried; these queries are dispatched to routines specific to the allocator managing the target area of memory. For example, querying a stack pointer dispatches to a stack walker; a global variable's address to a symbol table lookup; a malloc()'d heap chunk to a special 'index' maintained via link-time instrumentation of allocation functions.

3.2 Basic Design

Consider the following fictional example C library.

```
struct hw {
    int hello;
    float world;
};

struct hw hw_zero() {
    return (struct hw){};
}
struct hw *hw_p_zero() {
    return calloc(1, sizeof(struct hw));
}
void hw_print(struct hw arg) {
    printf("h%d,␣w%f\n", arg.hello, arg.world);
}
void hw_p_print(struct hw *arg) {
    printf("h%d,␣w%f\n", arg->hello, arg->world)
        ;
}
```

Listing 4. Fictional example C library

Our system is implemented as a CPython extension module which allows the example library to be used from Python as follows.

```
# Import the C library as if it were Python
import elflib.example as lib

# Manually allocate a struct hw
v0 = lib.hw()
v0.hello = 42
v0.world = 3.14
# or
v0 = lib.hw(42, 3.14)

# Get a struct hw from a C function
# Value vs reference is abstracted away in
    Python
v1 = lib.hw_zero() # Return by value
v2 = lib.hw_p_zero() # Return a reference

# Pass a struct to C function
lib.hw_p_print(v0) # Call by reference
lib.hw_print(v1) # Call by value
lib.hw_print(v2)

# Pass Python values to a C function
lib.hw_print((0,0.))
lib.hw_print({'hello'=0, 'world'=0.})

# Values can be passed structurally by matching
    field names
class Hw:
  def __init__(self,h,w):
    self.hello = h
    self.world = w
lib.hw_print(Hw(-1, float('nan')))
```

Listing 5. Python code calling functions from Listing 4

In summary, the following basic techniques enable this.

Shared objects as modules In Python, **import** is extended so that it can import native shared objects. Just as naming conventions are used to map to .py files in the filesystem, so they may now also locate shared objects. The client does not specify whether the module is native or in Python.

Functions, globals and types The imported shared object's code, data and types appear in the imported namespace. Python code may also directly instantiate native data types.

Proxying To allow CPython access to native objects, which do not conform to its header-based layout, we could either relax CPython's expectations or indirect through a proxy which does conform. To minimise invasive changes to CPython, for now we introduce proxy objects, which point to arbitrary native data. In CPython we create a distinct proxy class for each distinct composite type descriptor (struct, union or stack frame) provided by liballocs.

Base-type values as references Base-type values, like integers, booleans and floats are 'objects' in Python, but in C are bit-patterns in some *typed* context. Conceptually these are interchangeable if we view these bit-patterns as *references* to immutable and value-unique objects (e.g. there is a unique logical instance of the 'integer 2' object); they are interconverted accordingly.

Native structured data Composite types, such as structs and unions, have a proxy class allowing access to members. On field access, a new proxy may be created on demand to wrap the accessed value. It follows that proxies sometimes wrap interior pointers (see figure 1). To preserve Python semantics, assignment to a native pointer field stores a reference to the target object, or a null pointer if None is stored. Nested structs or arrays appear as 'references' that are read-only, to capture that the nested object instance is fixed in place.

Functions We define a CPython proxy class for native function objects. These proxies are Python-callable; when called, the proxy checks argument types, calls the underlying function and wraps the result in its Python form (proxy or native type). These proxies can be created on demand over any Python callable, creating closures as necessary.

Arrays Native arrays are proxied, behaving as Python arrays but with a fixed length. From Python they may be

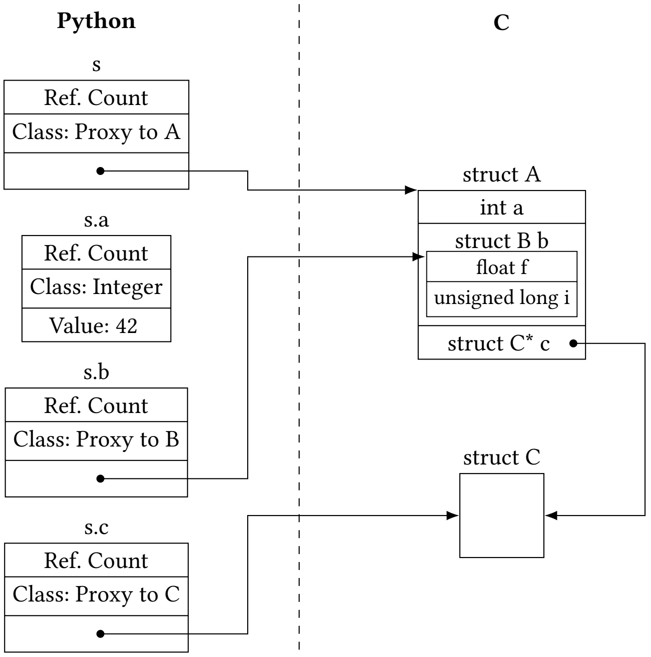

Figure 1. Illustration of Python proxies to composite structures, including interior pointers. Arrows represent pointers.

indexed into, and sliced. When a C library is loaded from Python, each data or function symbol is wrapped by a proxy of the appropriate type and is added to the resulting Python module.

4 Mediating Python and C Idioms

To provide a 'Pythonic' view of a typical C API, certain further techniques have proven necessary.

4.1 Containment Polymorphism

Some C interfaces rely on downcasting pointers, either from **void*** or from a smaller (contained) type, to a larger target type. Since casting is not a Python-language concept, we require some other approach. Consider a callback interface such as **void** on_click(**void** (*cb)(**void** *), **void** *arg); where the client-supplied callback must downcast arg from **void*** to a known type; or a 'base' structure type that is the first member of a number of larger containing types. (This pattern is used by the CPython VM itself for Python objects!) We resolve this simply by adopting Python's usual dynamic semantics: since run-time type information is available from liballocs, we query this at the point of proxy creation. (To avoid re-traversal of contained substructures, a further idiom also applies—see §4.3 below.)

4.2 Array Length

Arrays originating in C do not store bounds, but Pythonic behaviour demands raising an exception on indexing out of bounds. Again, dynamic information from liballocs enables this. Given a pointer into an array, querying liballocs yields the number of elements it contains, and this is recorded in the proxy.

4.3 Pointers and Arrays

C interfaces passing pointers are intentionally ambiguous as to whether the target is a single object or an array. Consider **struct** s *get_ref(); versus **struct** s *get_array();—the signatures are identical but the intention is different[2]. In Python code, we must resolve the ambiguity since the two cases imply different proxy classes. Our solution is the 'first element idiom': any array proxy also offers immediate access to the first element of that array, without an explicit index step. To handle containment polymorphism (mentioned above), this also applies to structures: a proxy to a containing structure also offers immediate access to any *contained* subobject beginning at the same address. More generally: for each type T, if the address of T is also the address of another smaller composite type T' (e.g. T's first element or member), then the members and operations of T' are exposed by the T proxy. This is arranged by definition of the proxy classes using CPython's inheritance mechanism. Fig. 2 shows an example. Importantly, this idiom preserves the property that any function 'returning $T*$' yields a proxy usable as a T. By contrast, under a naïve treatment, code whose signature says $T*$ but actually returns a pointer to S (containing a T as its first element) would require explicit access logic to get at the T, effectively undoing the intended polymorphism.

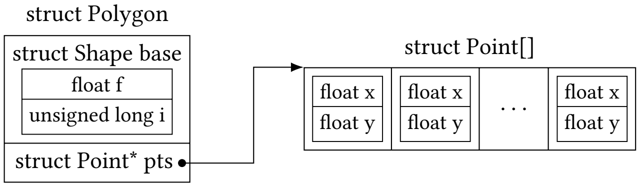

Figure 2. The 'first element idiom': accesses on the Polygon proxy can immediately access fields from Shape. Similarly, accesses on the array-of-Point proxy can access Point fields.

This allows the client code to manipulate a Polygon as if it were a Shape, or to use a pointer to an array to directly access its first element. Note that in the case of containment, the idiom is transitive: given an address returned by C code, any member beginning at that address, at any depth of nested containment, is accessible directly on the proxy. This can suffer from name clashes, e.g. if an object has the same field names as its nested structs. Although not currently implemented, this could be resolved by a name mangling convention defining alternative unambiguous aliases (roughly analogous to Java's **super**.x and similar).

[2] Pointer-to-array types do exist in C (**struct** s (*)[]) but are rarely used.

5 Garbage Collection

FFIs' biggest reason for existence is garbage collection (GC). GC implementation brings constraints on object layout, when objects may be deallocated or moved, and what must happen as pointers are passed around. In C, these are unconstrained and may vary by API. Whereas existing FFI systems typically push this complexity to the programmer, we seek to handle these automatically in the common case.

5.1 An Essential Conflict

C APIs embody some *policy* on *who* (caller or callee) is responsible for deletion of any object whose address passes over the interface, and also *when* and *how* this may be done—usually by calling a library routine like free(), but perhaps by stack re-use. Meanwhile, Python can be seen as enforcing a single policy, at the language level: objects are freed when they are no longer reachable. These two policies may disagree on whether an object is ready to be deleted.

FFIs push this conflict to the programmer, who must unnaturally 'keep a reference alive' to any garbage-collected object referenced from native code, and/or risk crashing the VM on the mistake of prematurely freeing a native-allocated object that the VM may yet reach.

5.2 Mediation of Policies

Our approach is instead to *mediate* the conflict from outside. This is achievable within liballocs, whose meta-level viewpoint already incorporates knowledge of the allocators coexisting in the process. We extend this to describe also the various *lifetime policies* to which an allocation may be subject. The object's allocator defines *how* it may be freed; its attached policies determine *when*. Logically, liballocs *intercepts* attempts to free an object before all policies agree, and *defers* the deallocation until later.

For example, an object allocated by malloc() but later passed to Python will carry metadata recording both *reference-counted* and *explicit-free* policies. If free() is called before CPython has signalled it as unreachable (by calling liballocs to remove the reference-counted policy), liballocs will arrange to defer the free() until this happens. Symmetrically, if the proxy's reference count hits zero first, the object will also not be freed until free() is called. Naturally, objects allocated other than by malloc() may have different (sets of) policies attached; for example, objects created from within Python, even if they are instances of data types defined in C, are subject only to the reference-counted policy.

We extended liballocs to implement this per-allocation policy metadata on malloc()-allocated objects. Each object may have one or more policies; only the two mentioned above are currently implemented. The per-object metadata amounts only to one bit per possible policy, since liballocs already knows which deallocation function may be used to free any given object (from tracking which allocators are managing which areas of the address space). These two policies broadly suffice in our scenario, provided that with C code we translate address-taken stack storage into a heap allocation which is freed on return, and disable (as permitted by POSIX) the ability of dlclose() to free 'static' storage.

Unlike notions of *ownership*, this 'policy mediation' model accepts that each unit of code—a native library, or a VM, say—has only *partial* knowledge of whether the object is ready to be freed. By contrast, ownership implies that exactly one party is responsible for deallocation; it fundamentally cannot resolve the conflict in all cases. Previous work has attempted to infer ownership by static analysis of escaped pointers, with partial success [14]. In general, however, while static optimisations are possible, we believe a dynamic approach to be more reliable.

5.3 Trapping Pointer Writes

CPython uses reference counting for automatic garbage collection. To provide natural Python semantics, our extension must also count references created or deleted in native code. We add a pointer write barrier, logically providing two upcalls to our Python VM: *addref* when a new pointer is written, and *delref* when an old reference is overwritten or erased.

This barrier degrades the performance of native code somewhat. Currently it is implemented naïvely as a source-to-source pass over C code, adding outcalls whenever a pointer is written to a global variable or to external memory, and also on memcpy() and realloc() operations (which may copy pointers). As an initial exploration of the spread of *compulsory slowdown* under this approach, i.e. the penalty suffered even when there is no Python code involved and no reference counts are maintained, Table 1 shows some benchmark data for native workloads instrumented by this write barrier calling out to a medium-short out-of-line code path (≈ 25 instructions), which is also called wordwise on the payload of realloc() and memcpy(). Only two of around ten workloads show substantial slowdown from adding the write barrier. A less naïve instrumentation would work at the binary level and/or with the help of alias analysis to identify writes whose referents are known not to escape to external code such as Python). We would expect this to show lower overheads.

Although a write barrier in a typical VM may amount to only a few instructions, we use this non-trivial code path because our approach must work for interior pointers and for diverse allocations. Unlike write barriers in a typical VM, it cannot assume a predictable object layout or the presence of a particular header. Therefore, inherently more work is involved in locating the metadata that the write barrier must update. Work remains to be done on exploring exactly what fast paths turn out to be optimal—for example, perhaps a fast check could rule out the interior pointer case, after which we could use prepended metadata much like an object header.

Table 1. Exploration of write barrier 'compulsory slowdown' on the SPEC CPU2006 benchmarks, showing uninstrumented baseline (gcc 4.9.2 + cil), against with-liballocs and additionally with the pointer write barrier calling a short out-of-line code path. Both percentages are relative to the baseline, run on an Intel Core i7-7700 4-core (8 logical) CPU at 3.60GHz (8MB cache), 16GB memory, Debian jessie chroot over Ubuntu 18.04 (Linux 4.15.0-55-generic #60). Medians of 5 runs shown; run-to-run variance was low in all cases, although buildwise variance is not characterised—this is evident in cases where allocs+wb is seen to perform slightly better than allocs. sphinx3 triggers a known performance bug in the March 2019 revision of liballocs used. perlbench suffers additional allocator instrumentation overhead. The failure of gcc is not yet debugged, although the write barrier has been successfully applied to that benchmark (in other, incomparable runs, approx 20% slowdown was seen).

bench	baseline/*s*	allocs/*s*	allocs+wb/*s*
bzip2	3.23	+0.30%	+4.1%
gobmk	10.3	+5.1 %	+41 %
h264ref	6.96	+8.4 %	+6.5%
hmmer	1.36	+8.9 %	+7.4%
lbm	1.21	+3.9 %	+2.2%
mcf	1.4	+4.0 %	+22 %
milc	3.07	+8.3 %	+6.6%
sjeng	2.27	+2.0 %	+2.9%
perlbench	3.31	+99 %	+98 %
sphinx3	0.771	+210 %	+190 %
gcc	0.652	×	×

However, interior pointers are frequently generated by C code, so it is not clear that this approach will perform well.

Our Python extension is not currently mature enough to demonstrate useful performance comparisons, although this is a work in progress.

5.4 Handling Pre-Existing References

Consider attaching a CPython proxy to an object to which references are already scattered through native data structures. To initialize the reference count, we would ideally count these references, requiring a potentially expensive heap traversal. To avoid this, we follow a pragmatic alternative: we undertake to track only references *created after* the GC policy was attached to an object. This is sufficient to ensure that any Python-visible object will not be reclaimed prematurely; it will be kept alive so long as *either* there exist references from Python-created objects, *or* there exist references *from native-created objects* that were written *while the object was proxied in Python.*

In effect, this leads to two kind of references in native data structures: 'strong' (counted) and 'raw' (uncounted). Raw references are those created when the object had only the explicit-deletion policy. They do not keep the object alive, and may become dangling, but are only created by native code. One surprising consequence is that C code such as `p->q = p->q;` may have the side-effect of converting a reference (in `*p`) from weak to strong (if `*p->q` is now proxied).

5.5 Realising the GC Policy Using Proxies

Reference counts live inside our proxy objects. When an object's reference count hits zero, the CPython GC lifetime policy is detached, and it is freed if there is no other attached policy. Otherwise, the object was created outside Python and an explicit free() is expected. Note that policies are not implied by an object's type: native-type objects created from Python start their life with only the CPython lifetime policy, so are managed automatically.

To prevent memory leaks on cycles, we modified CPython's cycle collector to follow all strong references during cycle detection involving a proxy object.

Our GC implementation must be able to identify quickly whether a target object is proxied. Currently we use two dictionaries: one from allocation base addresses of GC'd objects to the corresponding proxy object, and the other (rather like a 'remembered set') from strong references to the *proxy* of the pointed-to object. Currently these use stock CPython dictionaries, but since they strongly affect the performance of write-barriered workloads, work is needed on a custom structure which separates common-case fast paths.

5.6 Ongoing Work and Open Issues

Our handling of native stack frames has some limitations. Firstly, native local variables and other stack accesses are not reference counted, so approximations are necessary to bump the reference counts when objects are only stack-reachable. This would perhaps be best addressed by a periodic sweep rather than expensive fine-grained maintenance of counts. Alternatively, a more clever instrumentation pass could perhaps minimise count transitions without giving up fine-grained counts. Secondly, data stored in a C stack frame but accessible from Python code through closures currently cannot have its lifetime extended; to allow this it must first be promoted to heap storage.

Deferred deallocation is sometimes not enough to keep an object in a usable state, e.g. in the case of APIs having *finalization* separate from *deallocation.* Both finalization and deallocation should be deferred. Some native code comes with conventions for finalization (e.g. destructors in C++) so could be handled automatically. In C, no formal convention exists, so API-level annotation would be required. Since this could be useful debugging tools like valgrind, it could perhaps be packed into the library's debugging information, making it easily visible to liballocs.

Overall, in our context, reference counting appears a more difficult proposition than tracing collection. It is unclear whether tracing collection can be non-invasively adopted within CPython.

The proxy-based design stems from our choice not to disturb the internals of CPython. An extreme alternative to proxying would be to refactor the VM so that it makes no *a priori* assumptions about the layout of even 'its own' objects, treating them on par with 'foreign' objects. Roughly this means replacing '**struct** PyObject *' with '**void** *' (a change documented in earlier work [11]). Then, affected code would be reworked to make heavy use of liballocs's meta-level primitives and meta-allocator, instead of getting metadata from a VM-defined object header (hence PyObject type).

6 Related Work

Many works have followed a 'single shared virtual machine' paradigm [8, 13, 19], where garbage collection is handled by a common runtime, most recently with a shared compilation infrastructure that allows fast cross-language performance. They vary in the sophistication of their language mappings; in the Truffle-based work of Grimmer et al. [9] these back onto a fairly sophisticated operational metamodel, albeit remaining outside the end programmer's control. Since they rely on hosting native code unconventionally atop a language virtual machine, these approaches face compatibility issues on both native and (sometimes) higher-level code. Whereas these works have emphasised cross-language performance and sought to 'catch up' on compatibility [9, 15], we prefer the converse approach by using existing implementations and evolving a richer shared infrastructure underneath them.

A previous Python prototype exploited Dwarf information [11] for interoperability, but was otherwise a from-scratch rewrite, using the Boehm conservative collector [2].

CoLoRS [18] is similar in extending a shared heap between multiple high-level languages (e.g. Java and Python). However, it does not explicitly explore native code interop.

Many Lisp and Scheme implementations offer automated or unusual FFIs. To pick two examples, Larceny's layered FFI [12] is designed to preserve efficiency and interact well with the garbage collector. However, it does not extend collection to native code, and like Python's ctypes has limited data-level interoperability, building instead on a bare pointer primitive over which library-provided abstractions must be manually re-built in Scheme ('staying in the fun world'). A previous implementation of Scheme [17] provides a similar degree of wrapper-free integration as we do for Python, but lacking liballocs or similar mechanisms, does not support dynamism over native code.

The GNU implementation of Java [3] is noteworthy in allowing native libraries to be accessed by a linkage convention (CNI) rather than the usual FFI (JNI in Java's case), avoiding wrapper code. However, this does not integrate garbage collection into native code as we have done.

7 Conclusions and Future Work

We have described work towards a new approach in language interoperability between Python and C, which combines code instrumentation and additional run-time services to use native code conveniently within Python, including with dynamic semantics. Performance is already at a usable level but remains to be improved, and so far the extension interfaces of the CPython VM have sufficed, with no modifications necessary. Overall we have increased confidence that unnecessarily burdensome FFI approaches can be eliminated without reimplementing any of the languages concerned, and look forward to continuing efforts in this direction.

A Working Example (SDL2 & libpng)

The following example is executable by our current prototype, and illustrates the state of the work in progress.

```
from elflib import char
from elflib.libSDL2 import *
from elflib.libpng import *
from elflib.libc import fopen, clock
import atexit
import math
cstr = char.array

# == MACROS ==
PNG_LIBPNG_VER_STRING = cstr("1.6.37")
PNG_COLOR_TYPE_RGBA = 0x6
SDL_INIT_VIDEO = 0x20
SDL_WINDOWPOS_UNDEFINED = 0x1fff0000
SDL_PIXELFORMAT_RGBA32 = 0x16762004
SDL_TEXTUREACCESS_STATIC = 0
SDL_BLENDMODE_BLEND = 0x1
SDL_QUIT = 0x100
# == End of macros ==

png_file = fopen(cstr("dices.png"), cstr("rb"))
if png_file is None:
    raise FileNotFoundError("failed␣to␣open␣png␣
        file")

png_reader = png_create_read_struct(
    PNG_LIBPNG_VER_STRING, None, None, None)
if png_reader is None:
    raise RuntimeError("failed␣to␣create␣png␣
        read␣struct")

png_info = png_create_info_struct(png_reader)
if png_info is None:
    raise RuntimeError("failed␣to␣create␣png␣
        info␣struct")

png_init_io(png_reader, png_file)
png_read_info(png_reader, png_info)

width = png_get_image_width(png_reader, png_info
    )
height = png_get_image_height(png_reader,
    png_info)
row_bytes = png_get_rowbytes(png_reader,
    png_info)
```

```
color_type = ord(png_get_color_type(png_reader,
    png_info))
bit_depth = ord(png_get_bit_depth(png_reader,
    png_info))
if color_type != PNG_COLOR_TYPE_RGBA and
    bit_depth != 8:
    raise RuntimeError("png␣file␣do␣not␣have␣the
        ␣right␣image␣format")

imgbytes = unsigned_char_8.array(row_bytes *
    height)
imgrows = unsigned_char_8.ptr.array(height)
for row in range(height):
    imgrows[row] = imgbytes[row_bytes*row:
        row_bytes*(row+1)]

png_read_image(png_reader, imgrows)

png_read_end(png_reader, None)
png_destroy_read_struct(png_reader, png_info,
    None)

if SDL_Init(SDL_INIT_VIDEO) != 0:
    raise RuntimeError("failed␣to␣init␣SDL")
atexit.register(SDL_Quit)

win = SDL_CreateWindow(cstr("PNG␣display"),
      SDL_WINDOWPOS_UNDEFINED,
          SDL_WINDOWPOS_UNDEFINED, width,
          height, 0)
if win is None:
    raise RuntimeError("failed␣to␣create␣window"
        )

ren = SDL_CreateRenderer(win, -1, 0)
if ren is None:
    SDL_DestroyWindow(win)
    raise RuntimeError("failed␣to␣create␣
        renderer")

img = SDL_CreateTexture(ren,
    SDL_PIXELFORMAT_RGBA32,
    SDL_TEXTUREACCESS_STATIC, width, height)
SDL_UpdateTexture(img, None, imgbytes, row_bytes
    )
SDL_SetTextureBlendMode(img, SDL_BLENDMODE_BLEND
    )

while True:
    ev = SDL_Event()
    if (SDL_PollEvent(ev)):
        if ev.type == SDL_QUIT:
            break

    t = clock() / 100000
    r = int((1 + math.sin(t + 0/3 * math.pi)) /
        2 * 255)
    g = int((1 + math.sin(t + 2/3 * math.pi)) /
        2 * 255)
    b = int((1 + math.sin(t + 4/3 * math.pi)) /
        2 * 255)
    SDL_SetRenderDrawColor(ren, r, g, b, 255)
    SDL_RenderClear(ren)
    SDL_RenderCopy(ren, img, None, None)
    SDL_RenderPresent(ren)

SDL_DestroyTexture(img)
SDL_DestroyRenderer(ren)
SDL_DestroyWindow(win)
```

Listing 6. Working example calling function inside SDL2 and libpng from Python

Missing C features As the example shows, some corners of C such as macros and enums are not currently handled. Macro-defined constants and C-language enumerators are above redefined in Python code. These can be addressed using further metadata from the DWARF information. Strings are currently not managed very well: Python strings are immutable while a C function could modify an array of **char** (even when it is declared as **const**). Currently user code must create temporary char buffers when passed to C code. However, Python strings could safely be *externalized* (converted to a fixed representation), and passed to C code in immutability-preserving fashion using a read-only memory mapping—only the Python VM would retain access to the writable mapping. This requires translating any resulting segmentation faults into exceptions, a technique already employed by many VMs.

Supporting other languages We would like to support native code compiled from other languages (such as C++, Rust). Our source-to-source implementation of the pointer write barrier is currently C-specific but the idea extends to other languages, and could be done at the binary level. More generally, any language whose compilation yields DWARF metadata may be integrated into the existing liballocs system; the per-language effort is to describe its allocation functions (e.g. **operator new** in C++) to enable the necessary link-time instrumentation. In other cases such as OCaml, which use garbage collection and a high allocation rate, the link-time approach to intercepting allocation calls is not appropriate and a custom implementation of the meta-level protocol will be necessary. This also brings the challenge of integrating *multiple* garbage collectors, perhaps reified as distinct policies. This is an interesting topic for future work and entails the challenge of avoiding 'meta-cycles' between heaps.

B Introduction to liballocs

A fuller overview was given previously by [10]. Listing 7 shows a simplified view of liballocs's meta-level C API.

We can think of this API as a protocol which different allocators implement differently, but is collectively implemented *somehow* for every allocation in the process. Allocations are coherent, contiguous subdivisions of the state of a running program, having a base address and end address in memory. They constitute units of data meaningful (at some level) in

```
struct uniqtype;  // type descriptor
struct allocator; // heap, stack, static, etc
uniqtype *alloc_get_type(void *obj);
allocator *alloc_get_allocator(void *obj);
void *alloc_get_site(void *obj);
void *alloc_get_base(void *obj); // base address
void *alloc_get_limit(void *obj); // end address
Dl_info alloc_dladdr(void *obj);
```

Listing 7. A simplified liballocs process-wide metadata API

the program, where their "type" is a reified descriptor of that meaning, an instance of **struct** uniqtype. Allocation metadata also includes the site (instruction address) where allocation occurred.

The abstraction of allocations is a "common denominator" across all languages, runtimes and virtual machines, having both a machine-level view and (often) a direct relationship to the source program. However, they are lower-level than most languages' or VMs' notions of of "object": they intentionally lack any notion of behaviour, such as a system of messaging or method dispatch.

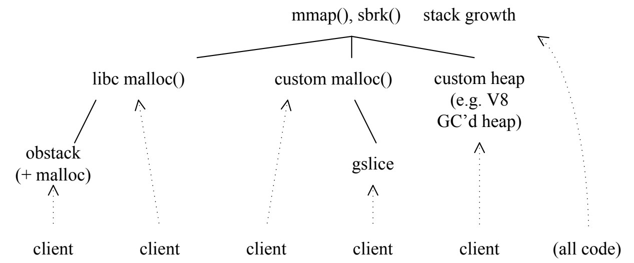

Figure 3. Allocator tree in a large C/C++/JavaScript program

Allocations form a hierarchy in a process's virtual address space. In a modern Unix, the bulk of a process's state is captured by its virtual memory, structured as a flat collection of *mappings*. Roughly, these mappings are the first level of the allocation tree. Mappings are parcelled out to user code via some arrangement of intermediate groupings which are also allocations: loaded segments, arenas, memory pools or slabs, stacks, and so on. These may nest further, for example when a heap allocator is itself implemented as a client of malloc(). The leaves of the tree are the units of state specified by the end programmer: local variables (roughly "stack" allocations), global variables ("static") and heap objects. Fig. 3 illustrates how allocators might be arranged hierarchically in a hypothetical Unix process.

Each allocation is also associated with an *allocator*, defined by a specific implementation of the meta-level API, This API can be thought of as a standard *meta-protocol* for allocators. Whereas allocators are intentionally unconstrained at the base level (they may expose whatever interfaces they choose, backed by any implementation), liballocs expects them to implement the meta-level API (or whatever subset of it their design allows).

At run time, liballocs maintains a collection of associative data structures. Dispatching queries to individual allocators is handled by a sparse array storing a 16-bit identifier for every page. This identifier is a key into the allocation tree. Only branch-level allocations, i.e. those with suballocations, need be stored explicitly in the tree. Leaf-level allocations are exclusively the concern of the given allocator and its implementation of the meta-level API.

As implemented in the wild, not all allocators maintain the desired type information or other metadata. Another role of liballocs is to provide utility code and hooking mechanisms for maintaining this metadata in side structures where necessary. For example, an associative data structure is used to record the type of each active malloc() chunk. Calls to malloc() and free(), and any other similar allocators that are declared to liballocs, are hooked by link-time instrumentation, so that per-chunk records can be maintained. Type information for malloc()-style heap allocations is recovered from a table keyed on the allocation site. For C code, a source-level analysis is provided, and run when building with the liballocs toolchain wrappers, which examines the use of **sizeof** to infer the type allocated.

Meanwhile, in the case of stack frames, type queries are answered without any additional per-allocation metadata. Instead, when walking the stack the active function's address is used as a key to look up the current frame layout. This is represented much as if it were a C-style struct data type, and is computed by postprocessing the debugging information of the containing shared object. Similarly, type information for "static" allocations, such as global variables, is looked up in an associative structure keyed by its offset within the containing dynamically linked object.

The data model representable in uniqtype descriptors follows that of Dwarf, which can be thought of as a lightly deduplicated 'superlanguage' comprising the data models of all Dwarf-described languages. To allow for details *within* an allocation to change dynamicaly, such as gaining or losing fields (e.g. a variant record) or varying length (a variable-length array), a single uniqtype can encode a bounded degree of per-allocation variability by providing a make_precise() function. This takes the object base address and (optionally) some additional program context, and returns a *dynamically precise* snapshot of the uniqtype. In this way, some details may be deferred until run time without spawning a distinct uniqtype for every possible case.

References

[1] DM Beazley. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*. 129–139.

[2] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.* 18, 9 (1988), 807–820.

https://doi.org/10.1002/spe.4380180902

[3] Per Bothner. 2003. Compiling Java with GCJ. *Linux Journal* (2003).

[4] Maurizio Cimadamore. 2018. State of the Isthmus. OpenJDK Panama design document; version 0.3 available at https://cr.openjdk.java.net/~mcimadamore/panama/panama-binder-v3.html as of 2019/8/3.

[5] Free Standards Group. 2017. *DWARF Debugging Information Format version 5*. Free Standards Group.

[6] Richard P. Gabriel. 1994. Lisp: Good News, Bad News, How to Win Big. *AI Expert* 6 (1994), 31–39.

[7] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. 1987. Shared Libraries in SunOS. In *Proceedings of the USENIX Summer Conference*. 375–390.

[8] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[9] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2724525.2728790

[10] Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

[11] Stephen Kell and Conrad Irwin. 2011. Virtual machines should be invisible. In *Proceedings of the compilation of the co-located workshops (SPLASH '11 Workshops)*. ACM, New York, NY, USA, 289–296. https://doi.org/10.1145/2095050.2095099

[12] F.S. Klock II. 2008. The Layers of Larceny's Foreign Function Interface. In *Proceedings of the Scheme Workshop*.

[13] E Meijer. 2002. Technical Overview of the Common Language Runtime. *language* 29 (2002), 7.

[14] Tristan Ravitch and Ben Liblit. 2013. Analyzing Memory Ownership Patterns in C Libraries. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2491894.2464162

[15] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[16] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. [n.d.]. A Survey of x86-64 Inline Assembly in C Programs. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*.

[17] J.R. Rose and H. Muller. 1992. Integrating the Scheme and C languages. In *Proceedings of the 1992 ACM conference on Lisp and functional programming*. ACM, 247–259.

[18] Michal Wegiel and Chandra Krintz. 2010. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*. ACM, New York, NY, USA, 223–240. https://doi.org/10.1145/1869459.1869479

[19] M. Weiser, A. Demers, and C. Hauser. 1989. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. ACM, New York, NY, USA, 114–122. https://doi.org/10.1145/74850.74862