A Foreign Function Interface between Gambit Scheme and CPython

MARC-ANDRÉ BÉLANGER, Université de Montréal, Canada
MARC FEELEY, Université de Montréal, Canada

Cross-language interoperability is a desirable feature of language implementations. We present such an integration between the Gambit Scheme implementation and the CPython Python implementation. Our work combines a syntactic interface relying on a custom parser to facilitate writing Python expressions directly in a Scheme program, as well as a low-level integration typical of other Foreign Function Interfaces (FFI). Our FFI uses a Foreign Procedure Call (FPC) mechanism which bridges the Gambit and CPython threading models. This work enables the use of Python packages from the Python Package Index (PyPI) from Gambit Scheme, opening a world of mature Python libraries to Scheme developers.

CCS Concepts: • **Software and its engineering → Interpreters**; **Runtime environments**; *Extensible languages*; *Multiparadigm languages*.

Additional Key Words and Phrases: Scheme, Python, Foreign Function Interface, PyPI, Modules

1 INTRODUCTION

The Scheme programming language, despite it being one of the oldest programming languages still in use, has a comparatively limited user base relative to younger languages such as JavaScript and Python. According to various surveys and metrics Python is one of, if not the most used programming language [1, 6, 17]. It is generally considered to be very approachable by non-programmers. A testament to its popularity is the number of Python libraries available through the Python Package Index (PyPI) [13], of which there are almost 400,000 as of August 2022. Researchers and practicioners in various fields can find a lot of software to work with in the Python ecosystem. Packages are available for domains such as computational social science [9], quantitative finance, numerical methods, 3D graphics, game development, network packet manipulation, machine learning or web development among many others. Furthermore, many packages are high quality and are actively maintained.

Scheme has not benefited from such large-scale adoption and has fewer easily available libraries. A cursory count of those offered by various implementations and package managers[1] arrives at a little over 3100 libraries. The number of Scheme libraries is orders of magnitude lower than what is available in Python. In Scheme, the gap between language extensions and libraries is filled by *Scheme Requests for Implementation* (SRFIs). However these are mostly concerned with the language itself and not any particular domain-specific libraries. While native Scheme libraries are desirable, making use of existing libraries — especially high quality, well maintained ones — is of great practical interest.

To interface with C libraries, Scheme implementations often offer a C Foreign Function Interface (FFI). Building upon our prior experience with interfacing Scheme with JavaScript, we present an FFI between Gambit Scheme and CPython, the canonical and most widely used Python implementation. Our work relies on a tight integration between the low-level CPython C API and Gambit Scheme as well as a high-level syntactic interface leveraging Gambit's Scheme Infix eXtension parser (SIX) to dynamically generate Python expressions. This combination offers programmers a versatile programming environment. In addition, developers can import Python libraries directly from the

[1]We counted libraries available for Gambit, Gerbil, Racket and Chicken as well as from Snow, Akku and Scheme Spheres.

Authors' addresses: Marc-André Bélanger, marc-andre.belanger@umontreal.ca, Université de Montréal, Montréal, Canada; Marc Feeley, feeley@iro.umontreal.ca, Université de Montréal, Montréal, Canada.

Scheme REPL. This means that instead of writing M FFI wrappers for M C libraries, we write one general FFI and gain access to all the libraries in the Python ecosystem, a significant advantage.

We hope this to be beneficial and interesting to Scheme programmers in general. We also wish to facilitate the adoption of Scheme by interested Python programmers who could bring along their favorite Python packages to get productive more quickly.

2 LOW-LEVEL INTERFACE

CPython's C API allows programmers to write extension modules for the language, or embed the CPython interpreter inside another program. We make use of CPython's C API through Gambit's *C interface*, which is the standard FFI for interfacing with C code. The C API may change across minor releases of CPython but we make use of the more stable parts and we have tested our FFI with versions 3.7 through 3.10.

In order to offer an efficient, flexible and robust FFI, many apects must be taken into consideration. Of great importance are the memory management models and the threading models of both implementations. The mapping of objects and their conversions have to be handled to both respect each Virtual Machine's (VM) memory management model as well as make object sharing as transparent as possible to simplify using the FFI. A low-level interface to Python can be useful for customization and some use-cases so it is important to expose it to the programmer in addition to the high-level interface that we discuss in Section 3.

2.1 Memory Management

A crucial aspect of handling Python objects from Scheme and *vice versa* is memory management. The CPython runtime uses reference counting to track object liveness, whereas the Gambit Scheme implementation uses a hybrid compacting GC.

A CPython object, which has the type `PyObject*` in C, can be referenced from Gambit Scheme using the general purpose Gambit Scheme `foreign` object, a cell containing a C pointer (in this specific case of type `PyObject*`) and a Scheme symbol *tag* indicating that `foreign` object's type which is used for run time type checking. The tag is typically `PyObject*` but a more precise tag is used when the specific Python type is know, such as `PyObject*/int` for a Python `int` object. For example, the CPython C API function `PyList_New` can be called from Scheme with `(PyList_New 5)` and this returns a Python 5 element list wrapped in the `foreign` object with tag `PyObject*/list` that prints using the format `#<PyObject*/list #2 0x102e63780>`. Each `foreign` object also has a pointer to a C function that is called when the Gambit Garbage Collector (GC) detects that this object is no longer reachable. For the `foreign` objects referring to Python objects the function simply calls `Py_DECREF` on the `PyObject*` pointer to let the CPython memory manager know that there is no longer a reference from the Scheme heap to this Python object. This balances the `Py_INCREF` that is performed when the `foreign` object is created.

Gambit Scheme objects can be referenced from CPython by using the CPython general purpose `Capsule` object, which is a cell containing a C pointer analogous to a Gambit `foreign` object. The C pointer references a Gambit `RC` object, which is a C structure containing a Scheme object reference and arbitrary other bytes not visited by the Gambit GC (in this specific use no other bytes are needed). `RC` objects have a reference count and are allocated with the Gambit runtime `___alloc_rc` C function which is functionally similar to the standard C `malloc` function except for the managed *data* slot containing a Scheme object reference. Our FFI defines the Python class `SchemeObject` to wrap the `Capsule` object and define a destructor as shown in Listing 1. By defining the `SchemeObject` class as a subclass of the `BaseException` class we allow Scheme objects to be used as exceptions on the Python side. When the CPython GC reclaims a `SchemeObject` object the destructor is called and this will decrement the reference count of the `RC` object using the Gambit

```python
class SchemeObject(BaseException):
  def __init__(self, obj_capsule):
    self.obj_capsule = obj_capsule
  def __del__(self):
    _pfpc.free(self.obj_capsule)
```

Listing 1. Implementation of the SchemeObject Python class.

runtime `___release_rc` C function. If this was the only remaining reference then the Scheme object referenced by the *data* slot no longer counts as a garbage collection root.

2.2 Threading

While both CPython and Gambit Scheme support concurrency through threads, their implementation models are different. Gambit Scheme supports concurrency using lightweight threads that can be created with the `make-thread` and `thread` procedures. In the default Gambit configuration these threads are time sliced on a single Operating System (OS) thread, but there is also an experimental support for multi-core execution that will eventually become the norm and that we want to plan for adequately in this work. Concurrency in CPython is achieved in multiple ways. The traditional way is with threads created with the `threading` module. Recent versions of CPython also support coroutines created by the `async def` keywords, and executed using the `await` keyword, and also generators (`yield` statement).

The Gambit runtime system for the C target supports multithreading as defined in SRFI 18 and 21. Scheme code is executed by a *processor* which is an abstraction of a physical processor typically mapped to an OS thread but that could also be mapped to a physical CPU core when running on bare metal. Gambit can be configured to use a single processor (the default) or multiple processors. Each processor runs a thread scheduler that maintains a queue of runnable Scheme threads on that processor and that time multiplexes their execution. Scheme threads are moved between processors to balance the load. The scheduler is implemented in Scheme by using first-class continuations to suspend and resume the execution of threads. To simplify the implementation of tail calls and continuations the Gambit runtime system manages a stack of Scheme continuation frames per processor that is separate from the C stack frames.

The consequence of this implementation is that there is not a one to one correspondence between Scheme threads and OS threads. On the other hand, the CPython implementation does have a one to one correspondence between Python threads and OS threads. For performance reasons, the Gambit runtime API, structures and memory manager have been designed so that only the processors (OS threads) that are created for that Gambit VM can execute Scheme code and participate in memory management and garbage collection. This leads to the following design issue concerning the threading models. A newly created Python thread cannot directly execute Scheme code by using Gambit's C interface, nor even allocate Scheme objects, because the thread does not correspond to one of the Gambit VM's processors. CPython's C API is more forgiving because the system has a Global Interpreter Lock (GIL) that must be acquired to perform most operations such as allocating and accessing Python objects. When the GIL is held by an OS thread, no other OS thread can call the C API. This is detrimental to Python thread concurrency but it does have the virtue of simplifying usage of the C API and our FFI makes use of this. Consequently there is an asymmetry: CPython's C API can be called from Scheme (by making use of the GIL), but for the most part the

Gambit runtime functions can't be called from Python threads. The solution to this problem and its impact on threading is explained in Section 2.3.3.

2.3 Mapping of Types

Inspired by our previous work with interfacing Scheme with JavaScript [8], we have chosen a general purpose mapping between Scheme and Python objects that is intuitive and practical. This mapping is implemented by the `(PyObject*->object python-object)` and `(object->PyObject* scheme-object)` procedures that are available to the programmer. Bidirectional mappings are favoured whenever possible so that an object round-trips to the same value (not eq? in general). For example, `(object->PyObject* 42)` returns a `foreign` object referring to the Python integer 42, and `(PyObject*->object (object->PyObject* 42))` returns 42. Our FFI makes use of this mapping for the automatic conversions of the high-level layer, as explained in Section 3.

Figure 1 lists the various conversions between Scheme and Python objects implemented by these procedures. Arrows indicate unidirectional or bidirectional mappings.

```
____Scheme____                  _______Python_______
#!void                   <=>    None
boolean #f/#t            <=>    bool False/True
fixnum  42               <=>    int 42
bignum  123...           <=>    int 123...
flonum  42.5             <=>    float 42.5
cpxnum  1.2+3.4i         <=>    complex 1.2+3.4j
ratnum  2/3              <=>    Fraction(2, 3)
()                       <=>    list []
list  (1 2 3)            <=>    list [1, 2, 3]
pair  (1 . 2)             =>    list [1, 2]
vector  #(1 2)           <=>    tuple (1, 2)
u8vector #u8(...)        <=>    bytes
s8vector #s8(...)         =>    bytes
string  "abc"            <=>    str "abc"
symbol  abc               =>    str "abc"
char    #\a               =>    int 97
table                    <=>    dict
foreign                  <=     foreign(python_obj)
(scheme scheme-obj)       =>    SchemeObject
procedure                <=>    function
                         <=     callable
                         <=     builtin_function_or_method
                         <=     method
                         <=     method_descriptor
```

Fig. 1. Scheme types and their mapping to Python types in Gambit's Python FFI.

2.3.1 *Simple objects and types.*

Bidirectional mappings are implemented for as many builtin types as possible. The Python `None` object is mapped to the Gambit `#!void` object which plays a similar role in Scheme. Python booleans `True` and `False` are mapped to `#t` and `#f`. The Python number types `int`, `float`, `complex`, and `Fraction` are mapped to Scheme exact integers (`fixnum` or `bignum`), `flonum`, `cpxnum`, and `ratnum`, respectively. Our FFI also bidirectionally maps the following types

from Python to Scheme: `str` to `string`, `tuple` to `vector`, `list` to `list`, `dict` to `table` and `bytes` to `u8vector`. The mapping from `tuple` to `vector` was chosen because the length of these objects can't change, whereas the mapping from Python `list` to Scheme `list` is both easy to remember and justified by the fact that their length is mutable, although not exactly in the same way.

Some types are only converted unidirectionally. Scheme `s8vectors` are mapped to Python `bytes` that are converted back to `u8vector` if returned to Scheme from Python. The Scheme types `symbol` and `char` are converted to Python `str` and `int`, respectively. We have chosen not to convert Gambit `keyword` objects because there is no useful mapping and it simplifies the handling of Python functions with keyword arguments.

2.3.2 Foreign objects and passthroughs. Foreign objects can be handled by either Scheme or CPython. From Scheme, foreign objects are CPython objects. From CPython, foreign objects are Scheme objects. The importance of allowing transparent handling of foreign objects can be appreciated from Figure 1. Since only a limited number of mappings are implemented and make practical sense, a passthrough mechanism is provided. Furthermore, some mappings are unidirectional. Programmers might thus prefer to store foreign objects to avoid round-trip conversion issues or simply because they do not need to recursively convert the object's contents. As such, we allow programmers to directly handle foreign objects, bypassing the usual conversion process.

Scheme foreigns in Python are converted to instances of the `SchemeObject` class. The creation of these passthrough objects is done by using the `scheme` Scheme procedure, which handles the creation of the `SchemeObject` instance and the `Capsule` and `RC` object it contains. The round-trip is garanteed, that is `(eq? obj (PyObject*->object (scheme obj)))` is true for any Scheme object `obj`.

The analogous passthrough operation for sending CPython objects unconverted to Scheme is the `foreign` Python function. The implementation creates a Python `Cell` object to contain the Python object. `Cell` objects are used internally by CPython to represent closures and we repurpose them for our needs instead of creating a new class. When `PyObject*->object` converts a reference to a `Cell` it simply extracts the cell's content by invoking the C API `PyCell_Get` function and no other conversion of the Python object is done (so it will be a `foreign` object with tag `PyObject*` or a more specific `PyObject*` subtype).

2.3.3 Foreign Procedure Call. Our FFI supports the bidirectional mapping of procedures. The threading model mismatch explained in Section 2.2 requires a special mechanism to pass control between Scheme and Python code. Because of the similarity with the Remote Procedure Call (RPC) approach which allows calls between two machines, we have called our approach the Foreign Procedure Call (FPC) mechanism because it allows calls to a foreign language.

When a given Scheme thread `S1` performs a call to a Python function, it will be executed in the context of some Python thread `P1`. For maximal usefulness it should be possible for the code executed on the Python side to itself execute a call to a Scheme procedure (in the Scheme thread `S2`) and again for this Scheme code to execute a call to a Python function (in the Python thread `P2`). This interleaving of languages is not only useful in the context of callbacks; it allows arbitrary mixing of any Scheme and Python functions transparently. We think that in the above example of interleaved execution it is natural for the programmer to expect thread `S2` to be the same as `S1` and similarly thread `P2` to be the same as `P1`. In other words, unless explicitly told otherwise, the system should use the same thread context during an interleaved execution. It should be no different than a sequence of nested function calls `F -> G -> H` that are all executed by the same thread when all the code is in a single language.

Our FFI realizes this model as follows. When a Scheme thread `S` performs a call to a Python function for the first time (for that Scheme thread), a new Python thread `P` is created. We say that

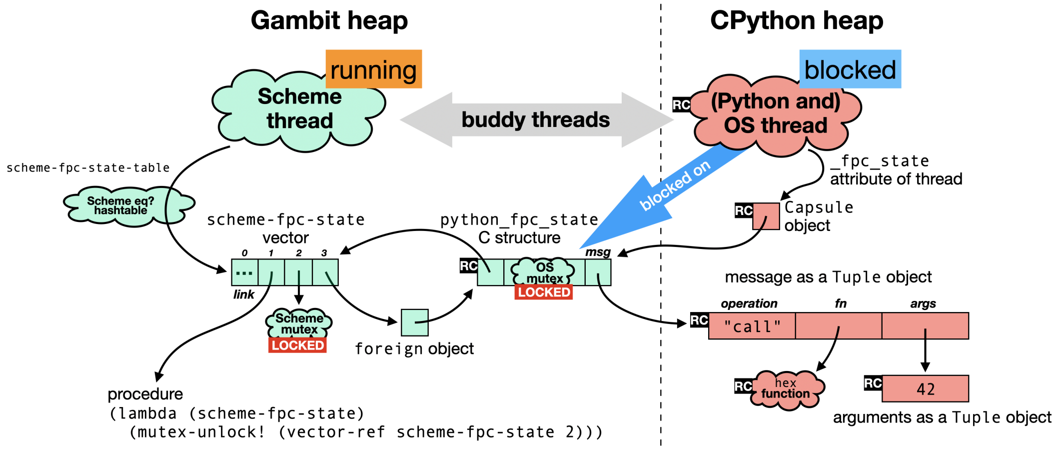

Fig. 2. The implementation of the FPC mechanism and the state when a call to the Python hex function with argument 42 is started from Scheme.

S and P are *buddy* threads. This link is maintained at run time so that both S and P can quickly find and communicate with their buddy thread. Thread P executes the Python function call that was requested by Scheme thread S. If this Python code then performs a call to a Scheme procedure, thread P's buddy (the Scheme thread S) will execute the call. Note that only one of a pair of buddy threads is executing at any point in time. The threads merely provide Scheme and Python execution contexts and no concurrency or parallelism. When execution switches from one language to the other the target thread resumes execution and the originating thread suspends it's execution waiting for the target thread to yield a result or, in the case of an interleaved execution, request the execution of another call. The buddy threads are thus executing as coroutines and when they block they expect to receive a message from the buddy indicating what to do next. This message will contain a function and list of arguments when a call needs to be performed. The message contains a result value when a foreign procedure call returns. There is also a need for a message indicating that a foreign exception was raised so that the originator of the call can be notified of this, causing the exception to be raised locally.

Figure 2 will be used to better explain our implementation of the FPC mechanism. It shows the buddy threads and the objects they use to coordinate and send messages. It is a snapshot at the moment the Scheme thread is calling the Python function `hex` with the argument 42. A pair of mutexes are used: one to block the Scheme thread (a Scheme mutex) and one to block the Python thread (an OS mutex). Both are in a locked state, and the Python thread is blocked on its mutex, trying to lock it. The Scheme thread has constructed a message (stored in the `msg` field of the `python_fpc_state` struct) containing the details of the call to perform (the function and the arguments). The message is always a Python object, a `tuple`, because both Scheme threads and Python threads can create Python objects with the CPython C API. Finally the Scheme thread unlocks the OS mutex and immediately blocks trying to lock the Scheme mutex which is currently in a locked state. Consequently the Python thread is now able to lock its mutex, read the message and take the appropriate action, which is to call `hex(42)`.

The roles are now reversed and it is the Python thread that constructs a *return* message indicating that the foreign call has ended and the value returned (the string `"0x2a"`). It then unlocks the Scheme mutex (as described below), and blocks trying to lock the OS mutex. The Scheme thread

```python
def SchemeProcedure(scheme_proc):
  def fun(*args, **kwargs):
    kw_keys = list(kwargs.keys())
    kw_vals = list(kwargs.values())
    return _pfpc_call(scheme_proc, args, kw_keys, kw_vals)
  return foreign(fun)
```

Listing 2. The definition of the SchemeProcedure Python function.

can now read the message and see that the foreign call has ended and it returns the value in the message. Had the Python thread wanted to perform a Scheme call rather than returning a result (i.e. an interleaved execution), the message would have been of type *call* rather than of type *return* and the Scheme thread would have performed a call rather than returning.

A tricky operation to implement is the unlocking of the Scheme mutex by the Python thread. As explained previously, most of the Gambit runtime functions can only be executed by OS threads that are Gambit VM processors. There is however a *procedural interrupt* mechanism that can be used by any OS thread to interrupt the execution of a processor and cause it to execute some Scheme code. This is the role of the Scheme vector `scheme_fpc_state` whose layout is compatible with an interrupt descriptor. When a procedural interrupt is raised using this descriptor it will cause a Gambit VM processor to execute the procedure it contains (at index 1), and this unlocks the Scheme mutex (at index 2).

The FPC mechanism involves more operations than a direct use of the CPython C API. The difference in performance is most acute when each call does little work, because the overhead of the FPC mechanism is a larger proportion of the total run time for the foreign call. On a modern computer running Linux, the highest cost we measured for the FPC mechanism, using the foreign call `sum([0])`, is 5x compared to a direct use of the CPython C API, which does not support threaded programs. The overhead becomes negligible when the foreign function does more work or consumes and produces more data that needs to be converted, what we consider to be the typical use-case for our FFI.

2.3.4 Procedures. Scheme procedures are converted to Python functions by the `SchemeProcedure` function as shown in Listing 2. Similarly, Python objects of type `function` (such as user-defined functions), `builtin_function_or_method` (such as `dir`), `method` (such as class instance methods) or `method_descriptor` (such as `datetime.date.ctime`) are converted into Scheme procedures. In addition, any callable object (i.e. objects for which `PyCallable_Check` returns true) are converted to Scheme closures. Both Gambit Scheme and Python support keyword arguments. These are supported bidirectionally in foreign calls.

To convert Scheme procedures to Python functions, `SchemeProcedure` takes as argument a `Capsule` holding a reference to a Scheme procedure. Using the FPC mechanism described in Section 2.3.3, calls to the resulting Python function are essentially evaluating the following code in Scheme:

```scheme
(object->PyObject* (apply fn (append *args **kwargs)))
```

When Python callables are converted to Scheme closures, the resulting closure will hold a reference to the Python callable. On every call, the closure performs an analysis of its arguments to properly construct the equivalent Python `*args` and `**kwargs`. These are sent as part of the `call`

```
> \2**8
256
> \secret_code=31416
> \[1, 2, [3, secret_code]]
(1 2 (3 31416))
> (define nb-dots 5)
> \(`nb-dots)*"."
"....."
> \print(42, end='xxx\n')
42xxx
> (list \1+2 +3)
(3 3)
```

Listing 3. Example SIX infix expressions.

message as described in Section 2.3.3. For performance reasons, we distinguish between simple calls and calls with keyword arguments.

3 SYNTACTIC INTERFACE

A distinctive feature of our FFI is its high-level layer. It is a syntactic interface supported by Gambit's Scheme Infix eXtension (SIX). We build upon our work interfacing JavaScript with Scheme [8] and extend the Gambit SIX infix parser to support a more Python-like syntax. Our motivations are similar: a syntactic interface reduces the barrier to exploration and interactive development between languages. It can allow for reusing code practically unchanged as well as prevent the need to map language features to the FFI host syntax, which could be cumbersome and hard to remember. Furthermore, the syntaxes of Python and Scheme are easily distinguished, which limits confusion about which context the code will be executed in.

By interfacing from Scheme to Python at the expression level, we are able to seamlessly incorporate Python code into Scheme programs. The basic premise of our extension is that programmers would rather write code than pass strings to an `eval` function. As such, Python expressions can be constructed by toggling a parser based on a hybrid grammar which supports most JavaScript and Python constructs in addition to the Scheme syntax. This toggling from prefix to infix is done in the spirit of Scheme's quasiquotation.

A backslash \ indicates to the Scheme reader that the following code should be parsed in infix notation. Such *foreign* infix code can be interspersed with Scheme expressions, indicated by using a backtick `` ` ``. This tells the parser to parse and evaluate the following code as a Scheme expression. Whitespace after a token is treated as an infix syntax terminator, except when it occurs between parentheses. This makes for a more visually pleasing nesting of infix and prefix syntaxes in the same program without requiring a semicolon at the end of an infix expression.

Infix expressions such as `\x+1` will be converted by the parser into the s-expression `(six.infix (six.x+y (six.identifier x) (six.literal 1)))`. The standard definition of the forms `six.infix`, `six.x+y`, `six.x=y`, etc confers a semantics close to C's, for example expanding `\x=10` into `(let ((temp 10)) (set! x temp) temp)`. Our FFI overrides the `six.infix` form so that the s-expression will be converted to an expression or statement in the language of interest represented as a string, which can then be evaluated by the CPython interpreter.

```
(define (##py-function-memoized descr)
  (let* ((x (unbox descr)))
    (if (string? x)
        (let ((host-fn (python-eval x)))
          (set-box! descr host-fn)
          host-fn)
        x)))
```

Listing 4. The definition of the ##py-function-memoized Scheme procedure.

Listing 3 shows examples of the infix syntax. Note that the parser understands Python list syntax, as shown on line 4, as well as normal and keyword arguments, as seen on line 9. These forms are quite often used in Python code. Tuple and list comprehension syntaxes are not supported by the SIX parser. Line 7 exemplifies the use of the ` character to evaluate Scheme expressions while in the infix syntax. In this case, the `six.infix` macro gets the value of the `nb-dots` Scheme variable. As for the significant whitespace, line 11 shows an expression that is parsed as a call to `list` with two arguments, the expression `\1+2` and the expression `+3`. Upon encountering the whitespace after `\1+2`, the reader switches back to an infix syntax. Also note that on line 9 single quotes can be used to delimit Python strings.

For efficiency reasons it is best not to call the Python `eval` function on each evaluation of a SIX form because multiple evaluations may be needed. For example the expression ``\print(1+`i)`` will be evaluated 10 times in the expression ``(for-each (lambda (i) \print(1+`i)) (iota 10))``. We get away with a single call to `eval` which evaluates a Python `lambda` form, in this case `lambda ___1: return print(1+___1)`, containing the code and then convert it to a Scheme procedure. This procedure is then called 10 times, once for each evaluation of ``\print(1+`i)``. The number of parameters of the generated `lambda` form is equal to the number of subexpressions prefixed with `. This is a form of memoization. It is achieved by storing the string representation of the `lambda` form in a Scheme box that is passed to the `##py-function-memoized` procedure, defined in Code Listing 4. On the first call, the string is evaluated by the CPython runtime. The box contents is then mutated with the corresponding CPython anonymous `function` object created by the `lambda`. This ensures that subsequent calls do not repeatedly invoke the Python compilation process on a string. The conversions of the parameters and result to and from Python are automatically performed, as explained in Section 2.3.

The handling of Python module imports is treated specially. We have extended the SIX parser to support the Python `import` and `from X import Y` statements. These are statements in the Python grammar so they must be passed to the `exec` function rather than `eval`. Our FFI currently supports the import grammar as it is in CPython. Module imports can thus be easily performed by invocations such as `\from my_module import a, b, c as d`.

Finally, note that executing Scheme code from Python is trivial using our FFI. One can easily export Scheme functions to Python with an assignment, for example ``\rev=`reverse`` allows Python code to call the Scheme `reverse` procedure under the name `rev`. Dynamic evaluation of Scheme code can be obtained by exporting a `scheme_eval` function that receives a string, parses it and calls the Scheme `eval` on it:

```
> \scheme_eval=`(lambda (s) (eval (call-with-input-string s read)))
> \scheme_eval("(iota 5 100)")
(100 101 102 103 104)
```

4 EXAMPLES

Let us now give an overview of our FFI in practice. We will first show a few ways to use Python modules from Scheme by putting the automatic conversions to good use. After these small examples, we will present an example with some performance considerations in mind.

4.1 Simple Module Import

Importing modules at the REPL is a matter of entering an `import` statement using the SIX syntax, as shown in Listing 5. Notice that the statement on line 1 looks identical to the equivalent Python version. The Scheme `cal` procedure is constructed by wrapping the Python `calendar` module's `month` method. As such, we can easily print the calendar's string representation as show on lines 4 to 10 of Listing 5.

```
1  > \import calendar
2  > (define (cal y m) \calendar.month(`y, `m))
3  > (display (cal 2022 9))
4     September 2022
5  Mo Tu We Th Fr Sa Su
6              1  2  3  4
7   5  6  7  8  9 10 11
8  12 13 14 15 16 17 18
9  19 20 21 22 23 24 25
10 26 27 28 29 30
```

Listing 5. Simple module import example.

CPython has access to the same `stdin` and `stdout` file descriptors as Gambit. We could have obtained the same result by calling `\print(calendar.month(2022, 9))`. One interesting feature of sharing console output and TTY is that the Python `help` procedure is usable within a Gambit REPL. For example, a call to `\help(hex)` will fill the terminal with the Python documentation for the builtin `hex` function. This help menu can be exited by pressing `q`, as one would do in a Python REPL.

4.2 Poor Man's Method Calls

Gambit Scheme has no builtin regular expressions module and does not implement SRFI 115. One could take the time to write an interface with the PCRE C library [11], for example, or port the implementation from SRFI 115. However, if we take the perspective of a Python programmer trying out Scheme, importing the Python `re` module would be an easy solution. Listing 6 shows one way to interface with the `re` module by implementing a poor man's method call.

First, the `python` Gambit module is imported, which sets up the CPython VM. The Python `re` module is imported on line 4. The syntactic differences highlight that the import statements on lines 1-2 and 4 do not target the same environment. While we could start using the module right away through the SIX interface, lines 6 through 11 define a converter for the `re.Pattern` type. The Python `re.Pattern` objects are compiled regular expressions, which can `match` and `search` strings for patterns. The `compile` procedure, defined on lines 13-15, produces `re.Pattern` objects, which are converted to the procedures returned by `re.Pattern-converter`. This offers a rather simple and *scheme-y* interface to these objects, as can be seen on line 19.

```
(import (_six python)
        (github.com/gambit/python))

\import re

(define (re.Pattern-converter obj)
  (lambda (attr . args)
    (case attr
      ((pattern) \(`obj).pattern)
      ((match)   \(`obj).match(`(car args)))
      ((search)  \(`obj).search(`(car args))))))

(define (compile pattern)
  (re.Pattern-converter
   \re.compile(`pattern)))

(define pat (compile "s....e"))

(define m (pat 'search "(sch3me)"))

\print(`m)
;; <re.Match object; span=(1, 7), match='sch3me'>
```

Listing 6. Code for the re.scm example.

4.3 Importing Packages From PyPI

As discussed in Section 2, our FFI manages its own virtual environment, which allows us to install packages from the PyPI without fear of conflicting with existing global Python installations. For convenience, our FFI exports the `pip-install` procedure, which takes a string as argument corresponding to a package name. Thus to install the `Flask` package from the PyPI, we can simply call `(pip-install "flask")`. Code Listing 7 shows how we can quickly build a web server using Scheme procedures as HTTP request handlers.

We can execute this program and send HTTP requests to `localhost:5000`, receiving the `"hello from v4.9.4-63-g18987297"` string as a response.

4.4 Plotting Data Using `matplotlib`

The `matplotlib` library is a well-known Python package available from the PyPI [5]. Its documentation contains a wide variety of examples, the first one of which concerns the creation of a bar chart [16]. Listing 8 shows a direct translation of the example code from Python to Gambit's infix syntax.

We can see the code looks very much like Python, and little effort was required to get it up and running. In our experience, modifications from Python to SIX infix syntax are minimal, which is a strength of our approach. The output of the code in Listing 8 is shown in Figure 3.

```
(import (_six python)
        (github.com/gambit/python))

\from flask import Flask

\app=Flask(__name__)

(define (home)
  (string-append
   "hello from "
   (system-version-string)))

\app.route("/")(`home)

(define flask-thread
  (thread
   (lambda ()
     (\app.run host: "127.0.0.1"
               port: 5000))))
```

Listing 7. A very simple Flask application in Scheme.

4.5 More Examples

The SIX interface allows for very simple experimentation. At the REPL, one can easily install and import packages such as `requests` to perform HTTP requests, `pandas` to perform data analysis, `pynetdicom` to create DICOM services, `beautifulsoup4` to parse HTML, or even `Boto3` to interact with Amazon S3. One can trivially get the weather in New York City by using the three lines of code of Listing 9. The call on line 3 produces the following output:

```
"A chance of rain showers before 5am. Mostly cloudy, with a low around 73.
 Northwest wind around 9 mph. Chance of precipitation is 30%."
```

5 RELATED WORK

Various other Scheme implementations offer integrations with CPython [10]. The Cyclone [7] and Chicken Scheme [2] implementations both have a Python FFI module. These modules integrate at the CPython C API level yet also offer useful macros to quickly interface with objects and methods. The `Darkart` Chez Scheme library [3] implements FFIs to various other language implementations, including CPython. It offers Scheme libraries that wrap Python libraries such as NumPy. However, the integration seems to be done by hand. The PyonR Python implementation is built on top of Racket [14, 15] and uses a hybrid approach with regards to Python modules. The implementation can access native Python modules through the CPython C API to susbtitute missing unimplemented PyonR modules. CLPython is an implementation of Python in Common Lisp [19] which allows interoperation between Common Lisp and Python, but not through the CPython C API. The `burgled-batteries` [12] Common Lisp package is an effort to implement a *deep* integration to CPython through its FFI. Our work shares similarities with the above projects, but distinguishes itself by the addition of a syntactic interface and a threaded FPC mechanism.

```
(import (_six python)
        (github.com/gambit/python))

\import matplotlib.pyplot as plt
\import numpy as np

(define N 5)
(define men-means (vector 20 35 30 35 -27))
(define women-means (vector 25 32 34 20 -25))
(define men-std (vector 2 3 4 1 2))
(define women-std (vector 3 5 2 3 3))
(define ind \np.arange(`N))
(define width 0.35)

\fig_ax=plt.subplots()
\fig=fig_ax[0]
\ax=fig_ax[1]

\p1=ax.bar(`ind, `men-means, `width, yerr=`men-std, label='Men')
\p2=ax.bar(`ind, `women-means, `width, bottom=`men-means,
                 yerr=`women-std, label='Women')

\ax.axhline(0, color='grey', linewidth=0.8)
\ax.set_ylabel('Scores')
\ax.set_title('Scores by group and gender')
\ax.set_xticks(`ind, labels=['G1', 'G2', 'G3', 'G4', 'G5'])
\ax.legend()

\ax.bar_label(p1, label_type='center')
\ax.bar_label(p2, label_type='center')
\ax.bar_label(p2)

\plt.show()
```

Listing 8. The code for the matplotlib.scm example.

```
\import requests
\r=requests.get("https://api.weather.gov/gridpoints/OKX/35,35/forecast")
(table-ref \r.json()["properties"]["periods"][0] "detailedForecast")
```

Listing 9. Getting weather forecasts from the weather.gov API.

6 CONCLUSION

We have presented the syntactic and low-level interfaces of Gambit Scheme's FFI to CPython. Development of this FFI required working at various levels of the Gambit and CPython implementations

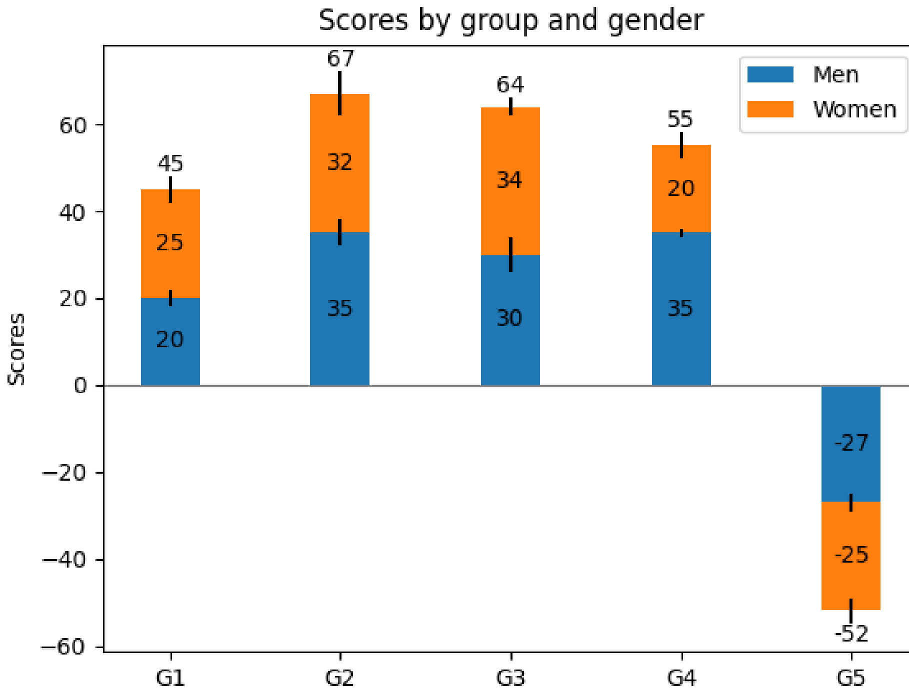

Fig. 3. A reproduction of the `matplotlib` *Bar Label Demo* code in Gambit Scheme.

from memory management to the implementation of the hybrid SIX infix parser. A defining feature of our FFI is its FPC design which allows seamless interoperation between the Gambit and Python threading models. These development efforts are largely offset by the ease with which we can now access Python packages from Gambit Scheme programs.

In future work, we plan to implement automatic generation of R7RS modules that wrap Python modules. An important unimplemented feature is the ability to have custom automatic conversions. We also intend to explore a higher-level interface to Python modules and classes using Scheme macros to generate interface code, as is done in Cyclone Scheme's `pyffi`. We believe this is especially interesting for large, complex modules such as `NumPy` [4] and `SciPy` [18]. With a careful design of the Scheme module interface and proper macros, we think it could be possible to offer a large amount of Python modules which would look and feel like they were written in pure Scheme. Finally, we are interested in reducing the interoperation cost to gain even more performance.

ACKNOWLEDGMENTS

The authors would like to thank Frédéric Hamel for his work on the `python-config.py` script and making everything work on Windows.

REFERENCES

[1] Stephen Cass. 2022. *Top Programming Languages 2022*. Retrieved 2022-09-02 from https://spectrum.ieee.org/top-programming-languages-2022 Section: Computing.

[2] Felix L. Winkelmann. 2022. *The CHICKEN User's Manual - The CHICKEN Scheme wiki.* Retrieved 2022-07-30 from http://wiki.call-cc.org/man/5/The%20User%27s%20Manual

[3] GitHub user guenchi. 2022. *Darkart: Chez Scheme's Foreign Library Interface.* Retrieved 2022-09-02 from https://github.com/guenchi/Darkart

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2 Publisher: Springer Science and Business Media LLC.

[5] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55 Publisher: IEEE COMPUTER SOC.

[6] Stack Exchange Inc. 2022. *Stack Overflow Developer Survey 2022.* Retrieved 2022-09-02 from https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022

[7] Justin Ethier. 2022. *Cyclone Scheme User Manual.* Retrieved 2022-07-30 from https://justinethier.github.io/cyclone/docs/User-Manual

[8] Marc-André Bélanger and Marc Feeley. 2021. A Scheme Foreign Function Interface to JavaScript Based on an Infix Extension. https://zenodo.org/record/4711425

[9] John McLevey. 2021. *Doing Computational Social Science: A Practical Introduction.* SAGE.

[10] Francesco Montanari. 2021. Scheme for scientific computing. In *Scheme and Functional Programming Workshop 2020.* 66.

[11] Philip Hazel. 2022. *PCRE - Perl Compatible Regular Expressions.* Retrieved 2022-07-30 from https://www.pcre.org/

[12] GitHub user pinterface. 2016. *burgled-batteries: A Common Lisp / Python Bridge.* Retrieved 2022-07-30 from https://github.com/pinterface/burgled-batteries

[13] Python Software Foundation. 2022. *PyPI · The Python Package Index.* Retrieved 2022-09-02 from https://pypi.org/

[14] Pedro Palma Ramos and António Menezes Leitão. 2014. An Implementation of Python for Racket.. In *European Lisp Symposium 2014.* 72–79.

[15] Pedro Palma Ramos and António Menezes Leitão. 2014. Reaching Python from Racket. In *Proceedings of ILC 2014 on 8th International Lisp Conference (ILC '14).* Association for Computing Machinery, New York, NY, USA, 32–38. https://doi.org/10.1145/2635648.2635660

[16] The Matplotlib development team. 2022. *Bar Label Demo — Matplotlib 3.5.2 documentation.* https://matplotlib.org/stable/gallery/lines_bars_and_markers/bar_label_demo.html

[17] TIOBE Software BV. 2022. *TIOBE Index.* Retrieved 2022-09-02 from https://www.tiobe.com/tiobe-index/

[18] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[19] Willem Broekema. 2022. *CLPython - an implementation of Python in Common Lisp.* Retrieved 2022-07-30 from https://clpython.common-lisp.dev/