Runtime Generated GObject Introspection Bindings for PyPy

Student: Christoph Reiter (#0731604)
Contact: reiter.christoph@gmail.com
Version: 1.0

Abstract

The scripting language Python has many successful applications as a glue language, bringing together low-level libraries in a scripting environment. In the past few years multiple alternative Python interpreter implementations like PyPy, IronPython and Jython have surfaced which focus on performance or improving integration with other language ecosystems. All of them suffer under the wide usage of Python extensions using the CPython C API, which is hard or impossible to implement for other VMs because it exposes many of the CPython implementation details such as reference counting. The resulting lack of usable extensions does not make these VMs a viable alternative for many applications and prevents their wider usage. This paper focuses on the PyPy interpreter and the CPython extension PyGObject, which provides Python interfaces for libraries based on the GObject system such as GTK+, GLib and GStreamer. PyGObject is widely used for application development for the GNOME desktop environment but also for cross platform applications and as GUI back-end for visualization libraries such as matplotlib. This paper covers how an alternative implementation of PyGObject, which also supports PyPy, would look like and how it compares the PyGObject. Further more this paper addresses the current lack of complete and sound documentation of the by PyGObject exposed API.

To help with various design decisions for an alternative PyGObject implementation this paper evaluates various approaches for interfacing with C libraries from PyPy and compares them with CPython. Based on these evaluations a prototype implementation called PGI is presented. PGI is compatible with PyGObject and produces bindings at runtime by generating Python code that uses ctypes/cffi to interface with the GObject based libraries. Based on this prototype implementation a documentation generator is presented which interfaces with the PGI extension to produce complete and sound API documentation for the generated interfaces. Since PGI is compatible with PyGObject the documentation is valid for both implementations.

While the prototype implementation is far from feature complete, the documentation generation it enables is now the default source of documentation for PyGObject users. The generated documentation helps users of PyGObject and PGI alike and reduces the initial hurdle for new users, especially those not familiar with C. Further more the presented evaluation of binding techniques can be helpful for similar projects, trying to provide performant bindings for PyPy.

Introduction

Looking at the various different Python interpreter implementations PyPy stands out as one of the most complete and compatible with CPython while providing significant performance improvements for certain workloads. For a subset of benchmarks PyPy uses to track its

performance over time it performs over 7 times faster than CPython 2 (see http://speed.pypy.org/). Compared to other implementations like Jython or IronPython it is more feature complete in regards to the language features and the standard library, as for example it is the only implementation containing a fully functional "ctypes" module for interacting with shared libraries. In some case PyPy even acts as a testing ground for new features of CPython. A proposed new dict (Python's hash table) implementation proposed by a Python developer (see https://mail.python.org/pipermail/python-dev/2012-December/123028.html) was first tested in PyPy (see https://bugs.python.org/issue27350)

While PyPy has various advantages, in practice it is not the implementation of choice for many applications. Some reasons being the worse startup performance, the unpredictability of performance due to the JIT and the used garbage collector strategy or the increased initial memory usage. One reason more is its lack of support for CPython C extensions, which provide Python interfaces for C libraries and are written in C against the public C API of CPython. One of those libraries, commonly used under Linux, is PyGObject, which provides a Python interface for libraries build on the GObject object system. One such library for example is GTK+, a cross platform GUI toolkit.

This paper looks at the various possibilities in how to create an PyGObject alternative which works with both CPython and Pypy, and for any implementation which does not provide full CPython C API support. We think that having a working alternative to PyGObject would help PyPy get wider usage as it currently lacks bindings for a cross platform GUI toolkit. But PyGObject not only provides access to GTK+, but also many other useful libraries such as glib, GStreamer and WebKitGTK.

Related Work

There exist two unfinished projects with a similar goal, to bring GObject library support to PyPy:

First a fork of PyGObject that partially implements the PyGObject API with ctypes. While this 1:1 translation from C code to ctypes would probably need less work to get 100% compatibility with PyGObject, the C code is designed around the idea that calls to shared libraries are cheap. This is not the case if ctypes is used and would lead to bad performance compared to PyGObject. The fork is currently unmaintained.

Secondly, the project pygir-ctypes exists with the goal to implement bindings for PyPy/CPython2.x/3.x that are not compatible with PyGObject. It does run minimal examples in its current form, but has shortcomings as it does not free any memory it allocates and the code is hard to extend. The project is also inactive.

Related to the goal of providing PyPy support for a GUI toolkit wxpython_cffi tries to implement a wxPython compatible interface. The project, which was part of Google Summer of Code, is no longer maintained and past implementation status reports can be found in the project associated blog.

Bridging the Gap Between Python and C

Integrating Python with other languages and runtimes is often required for reusing existing libraries, frameworks and systems written in another language. In case of CPython the common way to do this is to write an extension module using the CPython C API which links against CPython and the library one wants to interface with. Other Python interpreter implementations like Jython, using the JVM (Java virtual machine), and IronPython, using

the CLR (Common Language Runtime), in addition to interfacing with C libraries allow integration with their respective runtime. For the CPython C-API there also exist extension generators like Cython and SWIG which try to make writing extension modules easier using various approaches. Further more, Jython, IronPython, CPython and PyPy implement, to some extend, the ctypes FFI interface for calling C libraries using a Python interface. This chapter focuses only on integrating shared libraries written in C because this is what the GObject system is written in.

The interface of a shared library can be divided into two parts, the API and the ABI. The API (Application Programming Interface) is defined by the source code and the set of exported symbols and varies between different libraries and major library versions. The shared library itself doesn't include any API information besides the set of exported symbols and their names. The ABI (Application Binary Interface) is defined by the compiler and varies between different compilers and the architecture the compiler is building for. Some compilers also change this interface between compiler versions; for example GCC and Clang provide a stable C ABI across compiler versions, MSVC does not. When calling a function in a shared library, the API has to be considered for the function name, which type need to be passed to it and which type it returns. This information needs to be defined for each library by the respective library binding. The ABI has to be considered for how to place the passed values onto the stack, how to align them in memory and how to align values in structures. This information needs to be defined for each compiler and architecture and doesn't depend on the shared library used.

There currently exist two widely used approaches for writing Python bindings by using the provided API of the shared library.

The first approach is writing an extension using the CPython C-API. The created extension links against the shared library and calls the functions according to the API defined by it. Since all involved components, the CPython API provider, the extension module and the shared library are written in C and compiled by the same compiler for the same architecture, the ABI used is the same across all interfaces. The CPython C-API is provided by CPython and, to some extend, by PyPy.

The second approach makes use of a, so called, foreign function interface (FFI). The FFI provides functions for laying out values in memory and to call functions in shared libraries while complying with the ABI and the calling convention. One widely used implementation of such an interface is libffi for which bindings exist for CPython, OpenJDK and various Ruby implementations etc. In case of CPython and PyPy this interface can be accessed through the `ctypes` module included in the Python standard library or the external `cffi` module.

The following sections will focus on evaluating the different approaches for interfacing with C libraries for the Python interpreter implementations CPython and PyPy.

Benchmarking Setup and Limitations

All benchmarks and profiling runs performed in this paper were performed using a Intel i7-2640M CPU with 8 GB RAM using Debian Stretch as an operating system. The following software versions were used: Python 2.7.13, PyPy 5.6.0, cffi 1.9.0 and PGI 0.0.11.1. All the source code for generating the results and the plots should be accompanying this paper and can be modified or used to replicate the results.

All benchmarks presented in this paper focus on a small interface or a simple workload, so called micro benchmarks. They are not representative of real world workloads where

complex interactions between multiple workloads take place. For the implementation of the benchmarks care has been taken to make them reproducible, easy to run and easy to modify.

For comparing benchmark results of different Python interpreter implementations their implementation characteristics have to be taken into account. One such characteristic is the just in time (JIT) compiler which PyPy implements. The PyPy JIT compiler will optimized the program execution if it detects an often executed loop in the program. As this optimization process changes the result depending on at which point in time during the execution measurements are taken we want to either include or exclude it completely from the results.

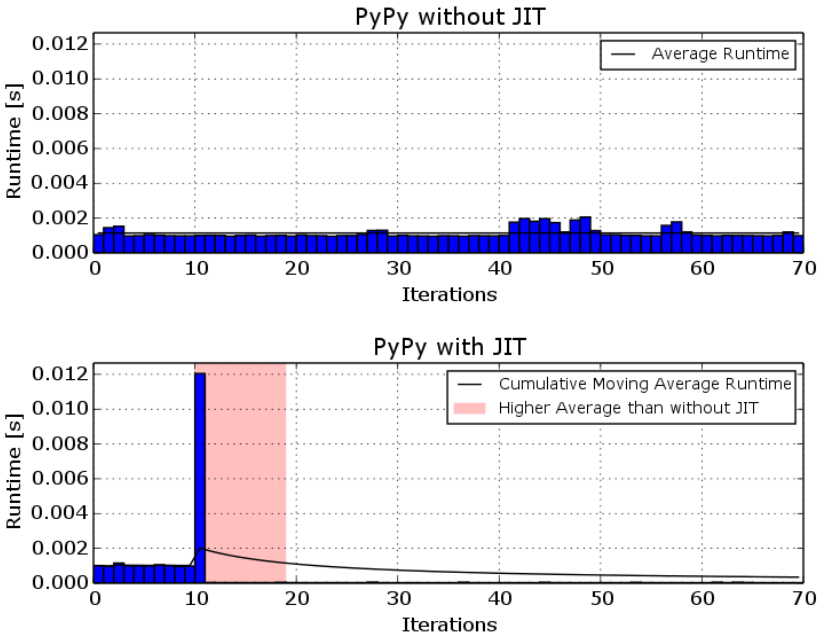


Fig. 1 Two bechmark runs for calls to *strlen* using the cffi library. Once with the JIT disabled and once with the JIT enabled.

The graph Fig. 1 illustrates the change of performance of calling *strlen* 100 times using the cffi library. The first plot shows the execution time with the JIT compiler disabled. The second plot shows the PyPy JIT detecting a, so called, “hot loop” around the 10th iteration and generating machine code for the trace which increases the runtime of that iteration. The red shaded area indicates the period where the average runtime is higher than without the JIT compiler showing that the optimization only pays of after around 15 iterations and can even reduce performance for short running programs. For long running programs the time until all the possible optimizations for a specific workload are done is called “warmup time”.

Because of these warmup effects the following benchmarks are grouped into two groups (1) runtime performance benchmarks and (2) startup benchmarks. For the runtime performance benchmarks the measurements start after the JIT compiler is finished optimizing the running program and only the optimized execution is recorded. This makes the results more representative of long running programs. The startup benchmarks record the execution of the program including all the warmup time.

Another difference between the CPython and PyPy is their memory management strategy. While CPython uses reference counts, PyPy uses a tracing garbage collector which periodically runs and can cause execution time peaks and extra variance in the results.

PyPy provides various options for tuning the GC and the JIT for a specific workload. For all benchmark runs in this paper no adjustments where made and the default settings where used.

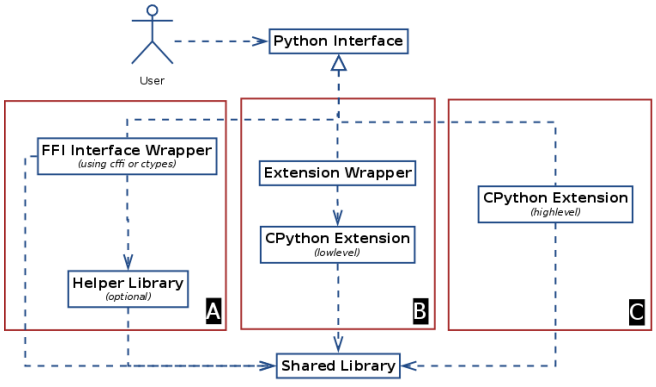
Library Wrapping Strategies

Python bindings for C libraries usually provide the following features on top of providing a Python interface:

- Hide memory management
- Validate input and output data and provide useful error messages with the goal that any misuse of the API results in a recoverable error and does not crash the program.
- Provide a Python class hierarchy for object oriented interfaces
- Translate various C error reporting schemes to Python exceptions. This includes among others, error codes, error flags in combination with a thread local error object or per call error objects.

These features can be implemented on different layers and the various strategies of laying out the implementation can be roughly divided in three groups, as visualized in diagram Fig. 2.

A: Implement the interface in Python only using a FFI module.
A/2: Optionally provide helper interfaces in a separate shared library linking to the wrapper shared library. Helpful for implementing things that are easier handled in C or for performance reasons.
B: Implement a minimal Python interface using the CPython C API and implement memory management, class hierarchies on top using Python.
C: Implement the whole interface using the CPython C API.



While the resulting API is in on all three cases the same there can be differences in performance, memory usage and code complexity.

There currently exist two implementations of application level foreign function interfaces for Python, ctypes and cffi. *ctypes* is the built-in FFI module shipped with the Python standard library and allows defining the C API in Python, loading libraries, calling functions and passing callbacks etc. It also exposes the CPython C API as a Python interface through the “pythonapi” attribute. This feature is optional and not supported under PyPy. *cffi* creates a Python interface by parsing C declarations. Compared to *ctypes* it allows for ABI-level compatibility by compiling a helper library, which makes the resulting module

platform depended like with C-API extensions, but not interpreter depended like CPython extensions.

And finally, to allow some degree of compatibility with CPython C extensions, PyPy includes the CPyExt subsystem which provides a compatibility layer for parts of the CPython C API. While it is not ABI compatible with CPython, it is partly API compatible and allows extensions to be recompiled against PyPy. CPyExt only implements a subset of the CPython API and not all implemented functions are 100% compatible with their CPython counterparts.

As an example, wrapping a simple C function `size_t noop_str(char*);` looks as follows in ctypes:

```
import ctypes

libnoop = ctypes.CDLL("libnoop.so")
noop_str = libnoop.noop_str
noop_str.argtypes = [ctypes.c_char_p]
noop_str.restype =  ctypes.c_size_t
```

while under cffi:

```
import cffi

ffi = cffi.FFI()
ffi.cdef("""
size_t noop_str(char*);
""")
libnoop = ffi.dlopen("libnoop.so")
noop_str = libnoop.noop_str
```

and a CPython extension module:

```
#include <Python.h>
#include <noop.h>

static PyObject* cwrapper_noop_str(PyObject* self, PyObject* args)
{
    char* raw_string;
    size_t out;

    if (!PyArg_ParseTuple(args, "s", &raw_string))
        return NULL;

    Py_BEGIN_ALLOW_THREADS;
    out = noop_str(raw_string);
    Py_END_ALLOW_THREADS;

    return PyInt_FromSize_t(out);
}

static PyMethodDef ctest_funcs[] = {
    {"noop_str", (PyCFunction)cwrapper_noop_str,
     METH_VARARGS, ""},
    {NULL}
};

void initcwrapper(void)
{
    Py_InitModule3("cwrapper", ctest_funcs, NULL);
}
```

```
from cwrapper import noop_str
```

In all three examples the global interpreter lock (GIL) gets released during the library call, which means other Python code can be executed concurrently until the function returns.

Performance Comparison of Python Extensions Interfaces

With multiple wrapping interface options available we need to look at how they compare performance wise under different scenarios. The goal is to get a better understanding of the

advantages and disadvantages of the various approaches so can make better design decisions for the planned Python bindings.

First we want to measure the overhead of calling functions in a shared library from Python. This should give us a base line of the minimal overhead to expect when building a Python interface.

For this benchmark three different functions with different argument types are tested. `bench_void()` takes and returns no arguments, `bench_str()` is similar to `strlen()` and takes a `char*` and returning a `size_t`, but compared to `strlen()` no length computation is performed. The third function `bench_double()` takes and returns a `double`, similar to the `fabs()` function of the C standard library. The functions are made accessible through C extension compiled against CPython and PyPy's CPyExt module as well as through ctypes and cffi.

The benchmark gets executed with both CPython and PyPy. In addition the benchmarks also contain a run with the JIT disabled to show the performance of code that is not executed often enough for the JIT to trigger as is the case with setup or startup code that gets only executed once on start or command line utilities where the requested work done is relatively small compared to startup or shutdown related work.

For measuring time differences the Python function `time.time()` was used which is implemented on Linux using the `gettimeofday()` system call which on x86 systems provides microsecond accuracy (Kerrisk, Michael. The Linux programming interface. No Starch Press, 2010., page 186).

Each function gets called 1000 times between measurements to reduce the dependency on the timer accuracy and reduce overhead of the measurement routine. To more accurately represent long running processes we try to not include the warmup period of the PyPy JIT compiler. For this to be the case, 6000 measurements were taken and the first 3000 ignored. This configuration was chosen based on the output of running PyPy with the `jit-summary` debug option enabled. Running PyPy with `PYPYLOG=jit-summary:stats.txt` will print the time PyPy spents in its tracing phase into the file `stats.txt`. The amount spend on tracing did not increase after 2000 runs up to 6000 runs, so it can be assumed that after 3000 runs the JIT compiler is finished optimizing the program execution, resulting in more representative results for long running processes. The results are as follows:

VM	FFI	Function	Mean [ms]	Stdev [ms]
CPython	C-API	bench_void	0.05060	0.00159
CPython	C-API	bench_str	0.10331	0.00222
CPython	C-API	bench_double	0.12493	0.00458
CPython	ctypes	bench_void	0.11490	0.00275
CPython	ctypes	bench_str	0.28395	0.00946
CPython	ctypes	bench_double	0.37051	0.04155
CPython	cffi	bench_void	0.09451	0.00439
CPython	cffi	bench_str	0.16475	0.00885
CPython	cffi	bench_double	0.14292	0.00468
PyPy	C-API	bench_void	0.17715	0.00527
PyPy	C-API	bench_str	0.82764	0.47006
PyPy	C-API	bench_double	0.98901	0.47044
PyPy	ctypes	bench_void	0.15611	0.00742

VM	FFI	Function	Mean [ms]	Stdev [ms]
PyPy	ctypes	bench_str	0.31300	0.01034
PyPy	ctypes	bench_double	0.22562	0.00796
PyPy	cffi	bench_void	0.01681	0.00100
PyPy	cffi	bench_str	0.04308	0.00162
PyPy	cffi	bench_double	0.01676	0.00084
PyPy-nojit	C-API	bench_void	0.45654	0.01375
PyPy-nojit	C-API	bench_str	1.21072	0.44854
PyPy-nojit	C-API	bench_double	1.44844	0.51510
PyPy-nojit	ctypes	bench_void	5.22192	0.17374
PyPy-nojit	ctypes	bench_str	10.14334	0.07547
PyPy-nojit	ctypes	bench_double	9.99481	0.40412
PyPy-nojit	cffi	bench_void	0.43314	0.06852
PyPy-nojit	cffi	bench_str	0.53657	0.00827
PyPy-nojit	cffi	bench_double	0.55958	0.00760

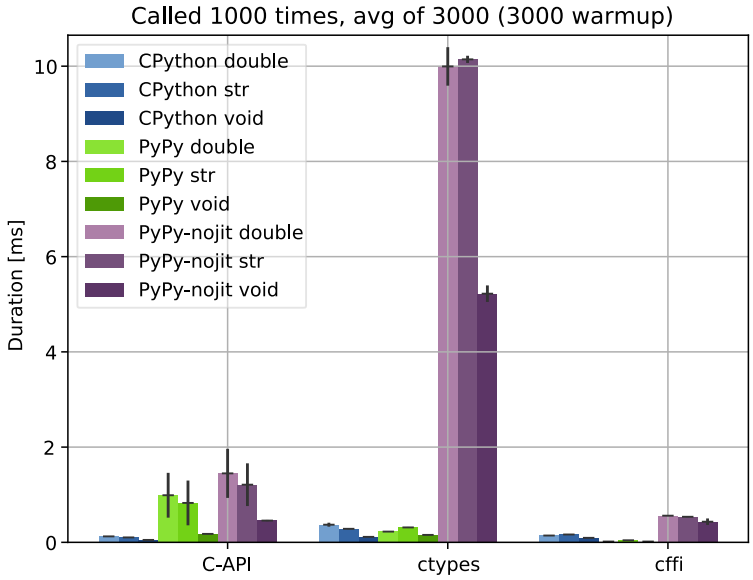


Fig. 3 Comparison of three APIs for accessing shared C libraries in Python using CPython, PyPy and PyPy with the JIT disabled.

Figure (Fig. 3) shows the result of calling strlen() and fabs() 1000 times. As can be seen, the C-API emulation module of PyPy performs significantly worse than the CPython one and the JIT is not able to improve much on that. As would be expected, PyPy with the JIT disabled performs the worst in all cases, with a significant outlier for ctypes, where the JIT is able to improve the call performance by a factor of ~30.

Given the performance of *CPyExt* and the fact that it does not cover all of the CPython API leads to the conclusion that it should not be used for new code and only for legacy code that is not performance sensitive.

Due to the bad performance of ctypes when run with PyPy and the JIT disabled it should be avoided for code which is only executed once or which the JIT is not able to optimize.

To get a better look at the better performing combinations see Fig. 4 for a zoomed in variant.

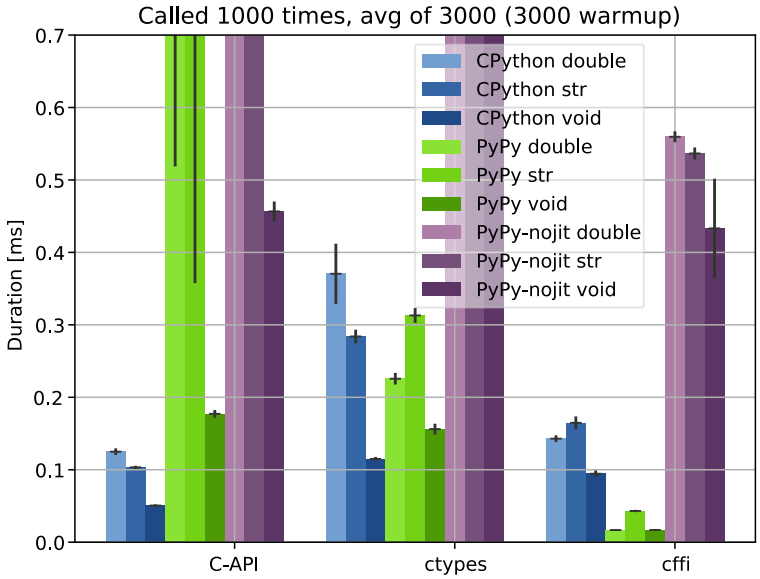


Fig. 4 Comparison of three APIs for accessing shared C libraries in Python using CPython, PyPy and PyPy with the JIT disabled (zoomed in)

If we take CPython with a C extension as a baseline for our comparison, we see that ctypes performs 2-3 times worse and cffi has similar performance when using CPython. PyPy + cffi on the other hand shows the best performance, with an improvement of the factor 3 for `bench_void()`.

If our target is having one implementation which performs well on both CPython and PyPy then using cffi is the clear choice, at least for these kinds of workloads.

Import Performance of FFI Libraries

Overhead of function calls is not everything that needs to be considered. The Python FFI modules have one downside compared to C extensions and that is that defining the C interface happens at runtime, so every time the program is run and the Python module imported. The work done at import time is also a workload which PyPy is not good at optimizing at, since it mostly concerns code which is only executed once in the lifetime of the program and the PyPy JIT only optimizes code which is run 100s of times.

To compare the import times of both Python FFI libraries, cffi and ctypes, we use the C library libgirepository, which provides an API for accessing GObject Introspection data for bindings and is one of the main APIs used in bindings like PyGObject and PGI. libgirepository's API (https://developer.gnome.org/gi/unstable/index.html) consists of opaque structures with inheritance, uses reference counting for memory management and GLib's GError API for error reporting.

The first comparison of both implementations concerns the import time, the time it takes for the API wrapper to load, after which it can be used by the user. The following benchmark shows the result of measuring the import of the API 1000 times. The result does not include the interpreter startup time.

VM	API	Mean [ms]	Stdev [ms]
PyPy	ctypes	114.84531	9.56864
CPython	ctypes	51.12451	4.32315
PyPy	cffi	627.61576	14.55936
CPython	cffi	298.70064	18.69073

The results show that the cffi based wrapper takes more than 5 times longer to import with both Python VM implementations as the ctypes based one. For analyzing where to most time is spent during import, the Python integrated `cProfile` profiling module can be used.

The results were generated using the following command: `python -m cProfile -s cumulative bench-cffilib.py`

```
         408826 function calls (402387 primitive calls) in 0.392 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        4    0.009    0.002    0.524    0.131 __init__.py:8(<module>)
        1    0.001    0.001    0.392    0.392 bench-cffilib.py:1(<module>)
       16    0.001    0.000    0.340    0.021 cparser.py:105(_parse)
    ....
    ....
```

The cProfile output shows various parameters of the execution for all executed functions. Sorting the output by cumulative time spend in a function shows that most time is spend in functions related to parsing C code. The C parsing routine `_parse()` takes about 87% of the total execution time in this particular run. This highlights one of the main differences how C API is defined in cffi compared to ctypes. While in ctypes this is done using Python code and can be made a lazy operation i.e. executed on first access, cffi infers the C API through parsing user provided C declarations. This in turn highlights a weakness of PyPy interpreter and its JIT compiler, in which it fails to optimize workloads which consist of lots of code only getting executed once.

Disabling the PyPy JIT shows one more problem. The following results compare the import times between PyPy with and without the JIT.

VM	API	JIT	Mean [ms]	Stdev [ms]
PyPy	cffi	on	627.61576	14.55936
PyPy	cffi	off	534.63665	17.22417

In case of the cffi API wrapper the import time decreases with the JIT compiler disabled. Using the `jit-summary` debug option shows that the JIT compiler takes around 250 ms to trace and optimize the execution but as the code does not get executed often enough the optimization process has a negative effect on the total execution time.

One work around for unnecessary JIT compiler activity would be a PyPy provided API to suspend the JIT compiler for a specific piece of code. This API could be used by the user in case code is known to be executed only once. PyPy currently only provides a way to globally disable the JIT at runtime which is not thread safe and thus needs a global view of the

program and can't be used in libraries and would need to be implemented by the applications themselves.

If we look back at the function call benchmark results we see that while cffi has less function call overhead compared to ctypes, especially in setup like scenarios where the JIT is not active, it is considerably slower at defining the C API definitions at runtime. This means for short lived applications, or applications where the initial response time and fast initial feedback matters, such as graphical user interfaces, ctypes might still be a better choice.

Performance of Error Handling and Input Validation

With the overall goal of providing an easy to use Python API for a shared library we also want to make sure that invalid usage of the API does not result in crashes or undefined behavior. We also want to be sure that passing too large integers will not result in silent truncation or overflow, and that invalid types passed to the API result in proper Python exception being raised. When using the Python FFI modules, where the final call happens on the Python layer, this validation also needs to be written in Python and not in C. In addition to input validation we also want to translate the various custom C error reporting schemes to Python exceptions.

To see how the performance of the validation layer compares when written in Python or in C we define the following C function called `overhead()` which covers a variety of common argument types and combinations:

```
int overhead(int32_t* list, size_t num, char* utf8, int* error);
```

- It returns an integer where the value indicates if an error occurred. If it returns a value other than 0 the function will write an error status code to the pointer `error`. This error reporting scheme is similar to the one used in GObject libraries with the GError API.
- The first two arguments are comprised of an integer array and a separate argument for the length of the array.
- The third argument takes an UTF-8 encoded, NULL terminated string.
- The last argument takes a pointer to where the error status code should be written to. If it is NULL no error code will be written.

The resulting Python API looks as follows:

```
def overhead(list_, text):
    """
    Args:
        list_ (List[int]): a list of integers
        text (unicode): a unicode text string
    Raises:
        Exception
        TypeError
        OverflowError
    """

    pass
```

- It just takes a list of integers and derives the length parameter it needs to forward to the C function from the list object. In case any of the contained integer objects can't be converted to a `int32_t` an `OverflowError` is raised.
- The `text` parameter can be a Python Unicode object and gets automatically encoded as UTF-8, as expected by the C function.
- In case the return value indicates an error `Exception` is raised.
- In case an object if unknown type is passed `TypeError` is raised.

The following benchmark compares the performance of calling the C function through cffi, with and without the extra Python validation layer. In addition the validation layer was also implemented in C using the CPython API. The results shows the time needed for 5000 calls with 6000 measurements and and the first 3000 ignored to not measure the JIT warmup effect.

VM	API	Type	Mean [ms]	Stdev [ms]
CPython	cffi	wrapped	10.60281	0.20072
CPython	cffi	bare	1.42904	0.13829
CPython	c-api	wrapped	1.99003	0.06758
PyPy	cffi	wrapped	0.74783	0.10501
PyPy	cffi	bare	0.48753	0.00877
PyPy	c-api	wrapped	20.76836	0.98269
PyPy-nojit	cffi	wrapped	20.89978	0.13806
PyPy-nojit	cffi	bare	3.42861	0.05461

While the validation does not add much overhead for PyPy with the JIT enabled, there is an increase in runtime for CPython by a factor of 4-5 and ~6 for PyPy with the JIT disabled. On CPython, the cffi call without validation is even slower than the C-API one with validation. This shows that reimplementing this API with FFI modules for it to be compatible with both CPython and PyPy will result in performance degradation when CPython is used.

CFFI Special Cases with PyPy Storage Strategies

While PyPy's JIT in combination with cffi can be faster than CPython when calling functions with simple integer arguments, it's still not able to beat CPython for passing lists of integers. In some cases the fact that all the marshaling is done in Python directly can have advantage, even over optimized C extensions. PyPy uses an optimization called "Storage Strategies" (Bolz, Carl Friedrich, Lukas Diekmann, and Laurence Tratt. "Storage strategies for collections in dynamically typed languages." ACM SIGPLAN Notices. Vol. 48. No. 10. ACM, 2013.) which change the implementation of a builtin data type on the fly depending on the modifications done on it. For example a list of integers that fits into the native integer data type doesn't get represented as a list of boxed values but directly as an continuous integer array. In case a value, not representable by that strategy, is added, the list changes to a generic list capable of holding all value types. One such strategy is the integer strategy for lists which stores the content of the list as a continuous array of integers in memory.

PyPy contains an optimization (see https://mail.python.org/pipermail/pypy-commit/2013-October/077864.html) where this allows simply copying the memory and passing it to the C function without the need to unbox and validated values. The following benchmarks shows the performance of calling a C function with the following signature and passing a list of 1000 integers without passing ownership to the function:

```
void int_list_args(int* list)
```

For both the cffi and C-API wrapper type and range checking is performed on all the list elements. One measurement represents 1000 such calls. The benchmark was run 1000 times to skip the PyPy JIT compiler warmup period and further 1000 times for recording results.

VM	API	Strategy	Mean [ms]	Stdev [ms]
CPython	C-API	None	3.37413	0.07061
CPython	cffi	None	13.82507	0.26181
PyPy	cffi	Integer	1.98485	0.20028
PyPy	cffi	Object	12.29639	0.63988

With the integer strategy, PyPy is faster than native CPython bindings. The optimization of the storage layout of collections not only improved the performance of Python code but also holds advantages for converting them to native C data types.

Resource Management and Garbage Collection

In some cases Python objects reference external resources, like for example file like objects which reference a file descriptor provided by the kernel or a image like object which references a memory block containing raw image data, allocated using the system allocator. By default the kernel limits the amount of file descriptors it gives out to processes to 1024 (see the output of `ulimit -n`). Similarly memory is limited, but the kernel tries hard to prevent it from running out by removing caches and unimportant processes. Thus external resource should be released right after they are no longer needed.

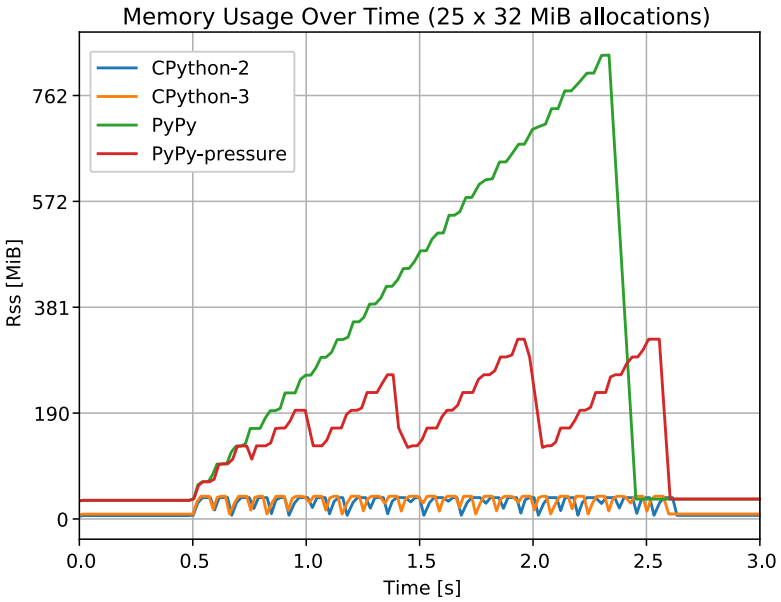
There are two ways how to handle the releasing of resources: (1) Expose it as a method of the object interface and require the user to manually release the resources when no longer used. This is currently the case with the `close()` method of file objects (2) Release the resource when the object is finalized by the Python garbage collector, either in the special `__del__` protocol method or by using a weak reference callback. In CPython this is often used as a fallback mechanism in case the explicit method was not used.

Tying the resource release to object finalization is not recommended because the Python language documentation states (see “reference-counting scheme” at https://docs.python.org/2/reference/datamodel.html) that there is no guarantee when and if objects are finalized. This missing guarantee can become a problem with PyPy when, for example, new file descriptors get created in a loop and closing them is deferred to the object finalization procedure, which might never happen as can be seen in the following demonstration:

```
>>>> while 1:
....     open("newfile", "wb")
....
[....]
<open file 'newfile', mode 'wb' at 0x00007f8c2d461920>
<open file 'newfile', mode 'wb' at 0x00007f8c2d4619a0>
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IOError: [Errno 24] Too many open files: 'newfile'
>>>>
```

In practice the finalization process in CPython is mostly predictable and deterministic. Only objects that are part of a reference cycle get not finalized right away when they are no longer reachable. Due to this fact some APIs don’t provide an explicit method for releasing resources. One disadvandage of explicit release is also that objects reach an invalid state and can no longer be used. Care has to be taken that all remaining references to the object are known and that nothing expects the object to be in a valid state. This can be seen, for example, with file like objects, where any operation on the object fails after it is marked closed and the file descriptor is no longer valid.

Memory Leak Example



Graph Fig. 5 shows an extreme example where the strategy of the PyPy GC leads to memory usage problems. The graph shows the memory usage of a program creating 25 objects where each object allocates 32 MiB and only frees them when the object is finalized. A short waiting period after object creation the last reference is deleted which makes the object unreachable. The memory usage measured represents the RSS (Resident Set Size) of the whole interpreter process. After the last allocation the GC is called explicitly to collect and finalize all remaining objects.

The CPython interpreters in both version 2 and 3 (“CPython-2”, “CPython-3”) finalize the objects right away and thus release the allocated memory. At no point in time is there more than one allocation alive. In case of PyPy (“PyPy” label) no memory gets freed until the final explicit invocation of the GC. The default mark-and-sweep GC of PyPy will only starts a collection run if it detects a certain amount of new allocations. Since the main memory allocation by the Python object is done through the system allocator, which the PyPy GC does not track, and the size of the tracked Python wrappers in memory is small, the GC will not start collecting.

For this problem PyPy exposes a `__pypy__.add_memory_pressure()` function which makes it possible to tell the PyPy GC about external allocations or deallocations. The passed number will be used to adjust the memory allocation threshold which influences the time of the next garbage collection cycle. The case where all the allocations and deallocations are passed to `add_memory_pressure` is shown as “PyPy-presure” in Fig. 5. A similar functionality also exists for other languages, for example GC.AddMemoryPressure() in ”.Net”.

Preventing Resource Leakage with PyPy

To make resource management with PyPy more predictable and behave more like CPython multiple strategies are possible:

Provide an API for manually releasing resources:

Gives the user more control for when to release resources but also puts the object in an invalid state, meaning more responsibility and error handling needed from the user.

Pass memory allocation information to the PyPy GC:
: For every allocation and deallocation with a known size, call `add_memory_pressure()`. Only applicable for memory allocations since the the PyPy GC strategy only works with memory allocations. In many cases the exact size of an allocated object is only known to the system allocator and not exposed through the API. The allocation size can also change over the lifetime of an object, for example when using lazy allocations or when resizing an allocation for a mutable object like an array.

Manually trigger a GC collection when an object becomes unreachable:
: In cases where objects which are known to allocate lots of external resources are no longer referenced by the user and don't provide a method for releasing those resouces the user can manually trigger a collection by calling `gc.collect`. For this to work the user has to be sure that all references to the object are gone so it can be finalized and has to consider the performance implications of the GC run.

Monitor resource usage manually and force a collection on large increases:
: Similar to what the PyPy GC does for its own tracked allocations, one can monitor resource usage and trigger a garbage collection in case a specific threshold is reached. In case of system memory this means regularly retrieving the process memory usage through either `mallinfo()` or the information exposed in the special `/proc` filesystem. The downsides are that these interfaces are platform specific and they have to be called regularly, but not too often to not degrade performance. On the other hand such a system could also be used for other types of resources like GPU textures.

  This strategy is currently used in the Javascript GObject bindings (https://wiki.gnome.org/Projects/Gjs) where increases of the process RSS will be monitored and a GC collection is forced once a threshold is reached (see https://git.gnome.org/browse/gjs/commit/?id=7aae32d1df14af87e9a31c785447b27012b64af9 for a relevant commit). A discussion regarding this strategy can be found in the archives of the gtk-devel-list mailing list: https://mail.gnome.org/archives/gtk-devel-list/2014-February/msg00027.html

  PyPy enhanced their garbage collector with version 2.2 (http://doc.pypy.org/en/latest/release-2.2.0.html) to make the major collection part incremental and allowing to split the collection task in multiple smaller sub tasks. A incremental collection can be invoked by "gc.collect(genneration=1)" and should be safer to call periodically without introducing long delays.

GObject Introspection: Metadata Extraction & Binding Integration

GObject is an object system for the C language. It makes it possible to write object oriented code with C, including inheritance, interface and includes its own type system on top of C. GObject has a focus on making it easy to use from other languages and language systems and bindings exist for Javascript, Perl, C++, C#, Python and many more.

In addition to making binding creation easy it also tries to minimize the needed maintenance work of bindings by moving all the needed logic into GObject and the C code and automate the binding creation as much as possible. This process is called GObject Introspection.

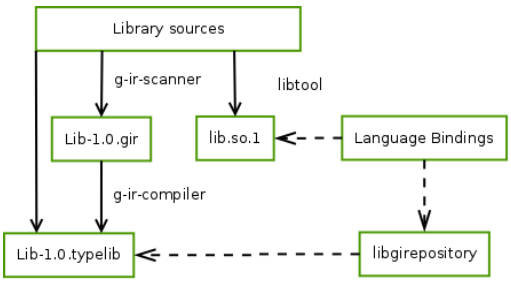


Fig. 6 GObject Introspection Process

The GObject Introspection process consists of two tools for extracting meta data from C code and the compiled library as well as a library for interfacing with the resulting meta data in bindings (Fig. 6).

The `g-ir-scanner` tool is responsible for extracting all the needed meta data from GObject libraries and creates a XML file containing all this information. These XML files are called GIR XML files or simply GIR (GObject Introspection Repository) files. `g-ir-scanner` extracts the needed information from the C header files and by introspecting the compiled library directly. These GIR files are usually generated during the normal build process of the library. For performance and memory efficiency reasons these XML files are further converted using `g-ir-compiled` to a binary representation called `typelib` which can be memory mapped and thus shared between different processes.

With `libgirepository` exists a C library which provides a stable API for the meta-data in the `typelib` files. Language bindings can use this library to gather all needed information about a library at runtime or build time. Each typelib defines a namespace identifier, which is a unique identifier for the library as well as a version to allow API incompatible changes. For example the GTK+ typelib has a namespace called `Gtk` and the versions `2.0` and `3.0`. A typelib can also have multiple dependencies, meaning it uses/references API from other typelibs and bindings need to load them as well.

The contained meta data describes such things as types and their relation, interfaces, object properties and signals, constants, closures etc. See the API documentation for more information.

To give an example on how a introspectable C function can look like:

```
/**
 * gtk_button_get_image:
 * @button: a #GtkButton
 *
 * Returns: (nullable) (transfer none): a #GtkWidget or %NULL in case
 *     there is no image
 *
 * Since: 2.6
 */
GtkWidget *
gtk_button_get_image (GtkButton *button)
{
...
```

“gtk_button_get_image” only contains additional annotations for the return type as everything else is either the default or inferred from the C signature. “nullable” means that the function can return a NULL pointer and “transfer none” means that the ownership of the passed object is not passed. Ownership in this case means the responsibility of decreasing the reference count, thus to caller has to increase the reference count if it wants to keep the object alive.

Using `g-ir-scanner` we can get the GIR XML:

```
<method name="get_image"
        c:identifier="gtk_button_get_image"
        version="2.6">
    <type name="Widget" c:type="GtkWidget*"/>
  </return-value>
  <parameters>
    <instance-parameter name="button" transfer-ownership="none">
      <doc xml:space="preserve">a #GtkButton</doc>
      <type name="Button" c:type="GtkButton*"/>
    </instance-parameter>
  </parameters>
</method>
```

And finally using `g-ir-compiled` we can generate a typelib file which can be used by PyGObject for example:

```
>>> from gi.repository import Gtk
>>> button = Gtk.Button(image=Gtk.Image())
>>> button.get_image()
<Gtk.Image object at 0x7f38a182c280 (GtkImage at 0x559883c68140)>
>>>
```

When new API gets added to the C library it automatically becomes usable from Python and other languages without the need to change the bindings themselves.

PyGObject

PyGObject is the official Python binding for GObject Introspection. For a basic tutorial on how to use PyGObject see the GTK+ tutorial.

To show how a PyGObject program might look like, the following code displays a window using GTK+ and terminates the program when the window is closed:

```
import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

window = Gtk.Window()
window.show()
window.connect("delete-event", Gtk.main_quit)
Gtk.main()
```

It imports the “Gtk” namespace in version “3.0”, creates a window and shows it, connects the event that gets triggered when the window is closed to a function stopping the main event loop and finally starts the event loop.

In addition to bindings, PyGObject also provides a set of module overrides. These consist of one Python module per namespace and can include new variables, new subclasses and setup code. For example, before GTK can be used, `Gtk.init_check` has to be called with the process arguments. `Gtk.init_check` will return a new filtered version of the argument list and a boolean which indicates if the initialization process was successful. Since this always needs to be done, it is included in the Gtk override module and executed when the user imports the Gtk module.

```
import sys

initialized, argv = Gtk.init_check(sys.argv)
sys.argv = list(argv)
if not initialized:
    RuntimeError("Gtk couldn't be initialized")
```

Python classes can define various special methods that get called by the interpreter to integrate them with Python language concepts. For example, the Gtk.TreeModel defines `__len__` which gets called by the built-in `len` function and returns the number of entries in

the model. By defining `__iter__` and returning an iterator object, it is possible to iterate over a model: `[row for row in model]`. These subclasses replace the original ones and every function returning a `Gtk.TreeModel` will return this new subclass instead.

```
class TreeModel(Gtk.TreeModel):

    def __len__(self):
        return self.iter_n_children(None)

    def __iter__(self):
        return TreeModelRowIter(self, self.get_iter_first())
```

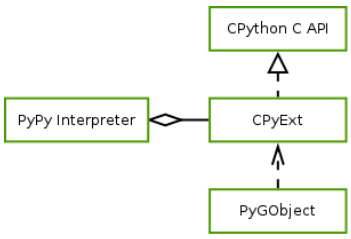
Another important use of override modules is to expose API that is not usable through GObject Introspection and implemented in PyGObject itself instead. To give an example of all the above features, the following script imports Gtk without any initialization code, creates a `Gtk.ListStore` with one row of the type `TYPE_UINT64` which is a constant defined in the overrides by calling `GObjectModule.type_from_name('guint64')`. Finally, since `Gtk.ListStore` is a subclass of the overridden `Gtk.TreeModel` which defines `__iter__`, it's possible to iterate over the containing values.

```
from gi.repository import Gtk, GObject

model = Gtk.ListStore(GObject.TYPE_UINT64)
model.append(row=[42])

for row in model:
    print row[0]
```

Since most of the overrides don't depend on PyGObject internals these overrides can be shared with our alternative library implementation and allow us to provide the same interfaces to the user.

Building with PyPy's C-API Compatibility Layer

Since PyPy has a compatibility layer for C extensions called CPyExt we tried to get PyGObject running (Fig. 7) with PyPy to have something to compare the new implementation with. This work resulted in the following bugs and patches being filed:



- https://bugs.pypy.org/issue1434
- https://bugzilla.gnome.org/show_bug.cgi?id=696646
- https://bugzilla.gnome.org/show_bug.cgi?id=696648
- https://bugzilla.gnome.org/show_bug.cgi?id=696650
- https://bitbucket.org/pypy/pypy/pull-request/146

These changes make it possible to compile PyGObject against PyPy and load it. Loading a library through PyGObject fails because of PyPy incompatibilities outlined in the above bug report. At the time of this writing it's not clear how much more work is needed to get PyGObject working with CPyExt and how it would compare performance wise. The needed work could mean changing PyGObject to use different API which is easier for PyPy to implement or just more commonly used and thus better supported, or to improve the compatibility of PyPy's emulation layer.

Implementation of PyPy Compatible GObject Bindings

Based on the results of the implemented benchmarks we designed a new PyGObject compatible library with support for PyPy called PGI. The primary goals of the new implementation were full compatibility with PyGObject on all Python versions and fast performance on both PyPy and CPython. While we could ignore CPython in regards to performance it is vital to make it also a viable alternative for CPython users. Firstly because most users use CPython which makes it more likely to find contributors and secondly to make it easier to switch to PyPy one library at a time and not force users to switch all libraries not compatible with PyPy at once. The following design decisions were made:

Cache everything needed for marshaling. On the first call of each method function all the state that needs to be computed is cached forever. This assumes that a program does only call a limited subset of all available methods and thus the caching does not result in large memory usage. This assumption was verified by modifying PyGObject to output all functions which are called and record the output when starting with a larger application (the audio player Quod Libet in version 3.8 was used). The startup of the application resulted in 354 different functions getting called.

Generate the marshaling code at runtime. To make it possible to optimize the marshaling code for each interpreter and Python version the code gets generated at runtime. This allows to specialize code for Python 3 or for CPython alone. As could be seen in the error handling and validation benchmark the Python marshaling code makes up a major part of each call when CPython is used and thus should be the focus of optimizations.

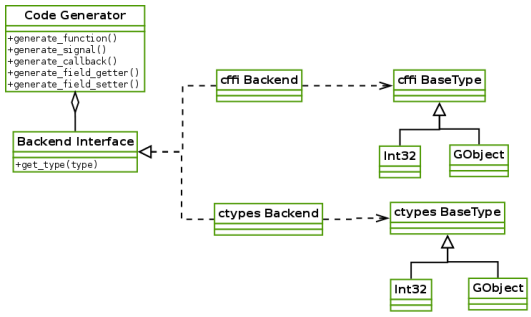


Fig. 8 PGI Overview Part 2/2

Have multiple code generation backends (Fig. 8) **.** PGI contains three code generation backends, one using ctypes, one using cffi and one that does not generate any code called NULL backend. They can be changed on a per function basis and allow PGI to use a different backend depending on the Python version and interpreter used which again allows for better performance. The NULL backend makes it possible to expose functions for which not everything is implemented in other backends and is used for documentation generation only, which will be described in a later section.

There currently exist two different versions of the Python language, Python 2, which only receives bugfix releases but is still the most popular one (see https://langui.sh/2016/12/09/data-driven-decisions/) and Python 3, which is actively developed and not fully compatible with Python 2.

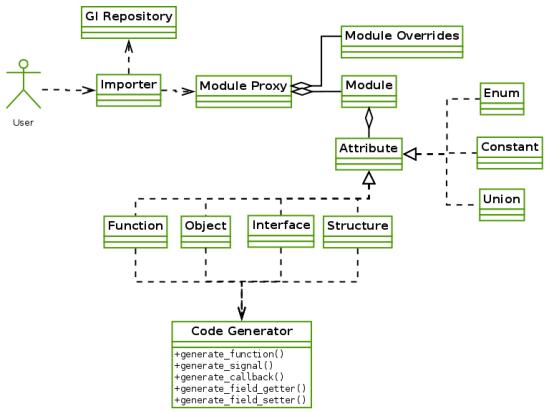


Fig. 9 PGI Overview Part 1/2

PGI's implementation can be grouped into three parts (Fig. 9)

- An API for accessing GLib and libgirepository to get all information needed for the binding generation. There currently exist two versions of this API in tree because, while cffi would be faster with PyPy, the import slowdown due to the parsing of C code, as mention in the benchmark section, increases the import time of the whole library too much. Thus the default API used in this part is ctypes at the moment.
- A code generation backend responsible for creating marshaling code for functions, methods, signals and callbacks.
- The rest which contains the dynamic import machinery, including the PyGObject API overrides and various implementations of functions which are not accessible through GObject Introspection.

PGI supports the following interpreter versions on Linux, Windows and macOS:

- CPython 2.7+ (Python 2)
- CPython 3.3+ (Python 3)
- PyPy 1.9+ (Python 2)
- PyPy3 2.4+ (Python 3)

Binding Example & Performance Evaluation

The following describes the process of writing a shared C library function and calling it in PyPy using PGI. For this we use a function already provided by the *gobject-introspection* test suite for performance testing. The method "regress_test_obj_torture_signature_0" takes an instance of *RegressTestObj* as the first argument and various in/out parameters with different basic types. The following code snippet shows the function definition and the needed annotations in the comment block above:

```
/**
 * regress_test_obj_torture_signature_0:
 * @obj: A #RegressTestObj
 * @x:
 * @y: (out):
 * @z: (out):
 * @foo:
 * @q: (out):
 * @m:
 *
 */
void
regress_test_obj_torture_signature_0 (RegressTestObj    *obj,
                                      int         x,
                                      double     *y,
                                      int        *z,
```

```
                                const char *foo,
                                int        *q,
                                guint       m)
```

Using the ctypes code generation backend PGI generates the following code which gets compiled to Python byte code at runtime and then cached on the “TestObj” class. The values shown in the comments before the function definition are contained in the function closure and are displayed there for debugging purposes.

```
# dependencies:
#   e19: <class 'pgi.codegen.funcgen.ReturnValue'>
#   e1: (<type 'str'>, <type 'unicode'>)
#   e14: <_FuncPtr object at 0x7f11f7bb5a10>
#   e3: <module '__builtin__' (built-in)>
#   e2: <module 'ctypes' from '/usr/lib/python2.7/ctypes/__init__.pyc'>
# backend: ctypes
def torture_signature_0(self, x, foo, m):
    '''torture_signature_0(x: int, foo: str, m: int) -> (y: float, z: int, q: int)'''

    # int32 type/value check
    if not e3.isinstance(x, e1):
        t4 = e3.int(x)
    else:
        raise e3.TypeError("torture_signature_0() argument 'x'(1): not a number")

    if not -2**31 <= t4 < 2**31:
        raise e3.OverflowError("torture_signature_0() argument 'x'(1): %r not in range" % t4)
    # new float
    t5 = e2.c_double()
    t6 = e2.byref(t5)
    t7 = e2.c_int32()
    t8 = e2.byref(t7)
    if e3.isinstance(foo, e3.unicode):
        t9 = foo.encode("utf-8")
    elif not isinstance(foo, e3.str):
        raise e3.TypeError("torture_signature_0() argument 'foo'(2): %r not a string" % foo)
    else:
        t9 = foo
    t10 = e2.c_int32()
    t11 = e2.byref(t10)
    # uint32 type/value check
    if not e3.isinstance(m, e1):
        t12 = e3.int(m)
    else:
        raise e3.TypeError("torture_signature_0() argument 'm'(3): not a number")

    if not 0 <= t12 < 2**32:
        raise e3.OverflowError("torture_signature_0() argument 'm'(3): %r not in range" % t12)
    # args: ['c_void_p', 'c_int', 'LP_c_double', 'LP_c_int', 'c_char_p', 'LP_c_int', 'c_uint']
    # ret: None
    t13 = self._obj
    t15 = e14(t13, t4, t6, t8, t9, t11, t12)
    t16 = t5.value
    t17 = t7.value
    t18 = t10.value
    return e19((t16, t17, t18))
```

This shows the function being called:

```
>>> from pgi.repository import Regress
>>> test_obj = Regress.TestObj()
>>> test_obj.torture_signature_0(5000, "foobar", 12345)
(y=5000.0, z=10000, q=12351)
```

To compare the performance we call it 1000 times and do 6000 benchmark runs while ignoring the first 3000 to exclude any JIT warmup effects:

VM	Library	Mean [ms]	Stdev [ms]
CPython	PyGObject	0.91911	0.03639
CPython	PGI	6.38835	0.62356
PyPy	PGI	2.04760	0.93081

Our implementation is more than two times slower with PyPy and more than 6 times slower with CPython than PyGObject. For this particular function there are still some optimizations possible, but it’s unclear how much they would affect the performance:

- Some methods on closure objects are looked up multiple times. As this is not optimized on CPython the methods could be cached in the closure as well.
- Not all type checking is needed. For many cases ctypes will raise if a wrong type is passed based on the set function signature. To still get a proper error messages pointing to the real cause including argument names the real cause could be computed in an exception handler.
- The code generated uses ctypes even though PGI prefers the cffi backend since the cffi backend does not implementing return values through pointers. Based on the benchmark comparing the FFI modules we can assume that cffi would be faster, especially when used under PyPy.

Status & Working Examples

While PGI is far from feature complete it can run various small example applications. The git repository (https://github.com/lazka/pgi) contains an examples directory with various working examples using GTK+, Clutter and Cairo. For example the small application window with menu bars and context menus as see in Fig. 10.

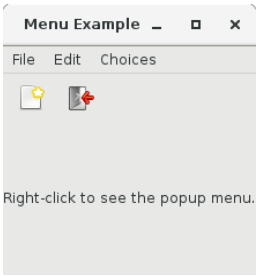


Fig. 10 An example program being run with PyPy and PGI. The code can be found in `examples/gtk/menu_example.py`

The examples directory also contains code for using PGI as a matplotlib backend. The matplotlib plotting library depends on the CPython API, numpy, GTK+ 3 and pycairo. Using pgi instead of PyGObject, cairocffi instead of pycairo and CPyExt to emulate the CPython API it is possible to display and interact with simple plots as can be seen in Fig. 11.

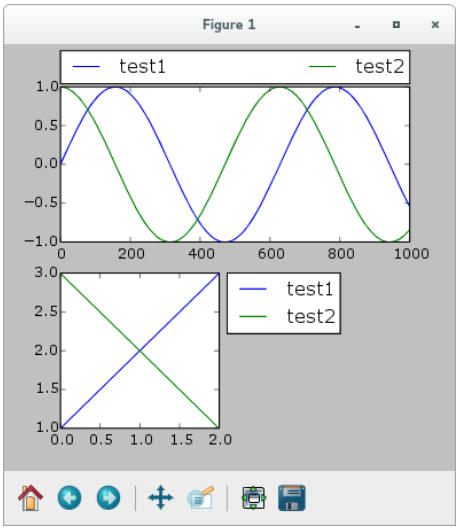


Fig. 11 Matplotlib 1.4 compiled using CPyExt using PGI and cairocffi as a backend

Due to the incompleteness of PGI it is not yet feasible to create a full featured GTK+ based user interfaces using PGI. A possible alternative was explored with the pypui framework which uses PGI only to open a web application and serialize commands between the Javascript and Python layers. An example application can be seen in Fig. 12.

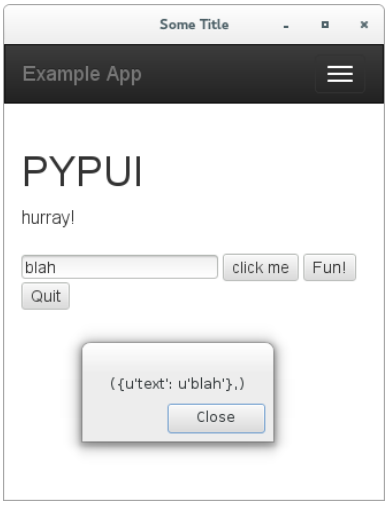


Fig. 12 A simple demo application which passes data from the WebKit UI to PyPy which in turn displays it in a GtkDialog window.

Automated Documentation Generation

One of the major drawbacks of the new GObject Introspection system compared to old static bindings like pygtk from a developer perspective is the lack of documentation. The importance of good documentation became obvious while watching the PyGObject support channels during the implementation of PGI as many of the questions asked there were regarding the Python API and missing documentation.

The automation the documentation generation was already a goal of the GObject Introspection project as the annotation scanner also extracts the accompanying comments and puts them into the GIR XML file. On the other hand they don't get included into the typelib file and are thus not accessible through the libgirepository API at runtime. The following XML snippet shows the meta data of the `gtk_button_set_label()` method including documentation:

```
<method name="get_label" c:identifier="gtk_button_get_label">
  <doc xml:whitespace="preserve">Fetches the text from the label of the button, as set by
    gtk_button_set_label(). If the label text has not
    been set the return value will be %NULL. This will be the
    case if you create an empty button with gtk_button_new() to
    use as a container.

    by the widget and must not be modified or freed.</doc>
  <return-value transfer-ownership="none">
    <doc xml:whitespace="preserve">The text of the label widget. This string is owned</doc>
    <type name="utf8" c:type="gchar*"/>
  </return-value>
</method>
```

For automatic documentation generation the GI project already contains a tool, called "g-ir-doctool", which uses the XML data to generate HTML documentation for Python, but the results are not complete and error-prone and thus the user still has to look up documentation of the C library and manually verify the interfaces with the Python interpreter. This state is the result of various shortcomings of the used generation

approach. First the documentation syntax used in the C code is a form of Markdown mixed with Docbook XML. The main consumer for the documentation and the implementation is “gtk-doc”, which is written in Perl. To get the same output as “gtk-doc” one has to closely follow its implementation and “g-ir-doctool” does not implement a compatible parser to get the same output. Furthermore not all the needed information is included in the used GIR XML. Some information like property descriptions and property default values can only be retrieved from the C code or at runtime through the GObject API. And finally in some cases PyGObject changes the exposed Python interfaces, either because the function is not usable through GObject Introspection and is implemented through the CPython C API or in case the API was extended to make it easier to use or for backwards compatibility.

To produce sound and complete documentation we’ve looked into a different approach by using the Python bindings as main source of the documentation.

PGI-Docgen - Documentation Through Runtime Introspection

To fix all the previously mentioned problems and provide a complete and sound documentation for PyGObject developers, PGI-Docgen was created. Instead of using the GIR XML as input, PGI-Docgen introspects the API exposed by PGI and only uses the GIR XML to pull in the documentation text. Compared to PyGObject, PGI exposes additional interfaces needed for documentation generation (see Fig. 13 for an overview). This has the advantage that the documentation always shows the real interfaces of PGI and in cases where PGI adjusts the API, for example through API overrides, the documentation reflects those changes. Since the introspection happens at runtime all the information available to the developer can be retrieved as well, like default property values. Despite PGI not being feature complete its NULL backend allows it to expose all possible APIs even in cases where it’s missing features for implementing them.

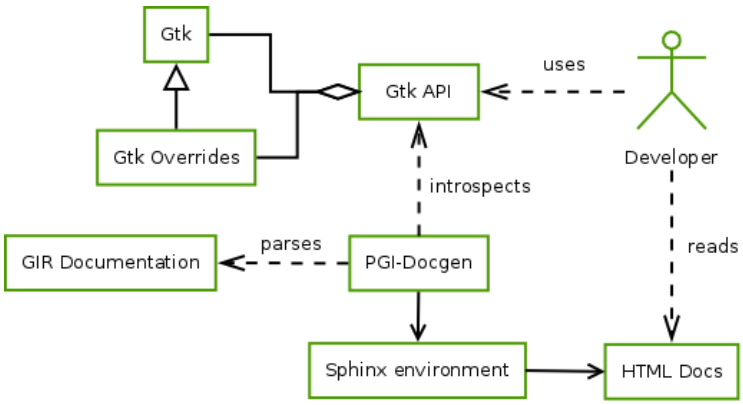


Fig. 13 Documentation Generation Process

To convert the retrieved information to HTML the Sphinx documentation generator is used which requires its input to be reST (reStructred Text). To match the formatting and output of “gtk-doc”, PGI-Docgen includes a Python port of the “gtk-doc” parsing code which converts the code comments to DocBook XML. PGI-Docgen then converts the DocBook XML to reST (Restructred Text) and parses the non-formatting related text for references.

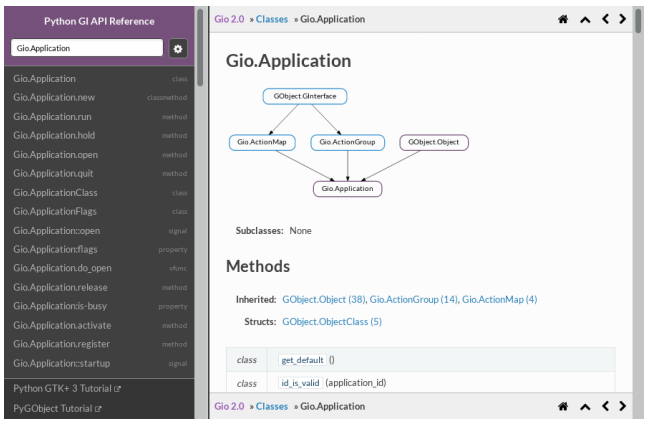


Fig. 14 Screenshot of the generated documentation describing the *Gio.Application* class.

The resulting documentation can be viewed at https://lazka.github.io/pgi-docs/#Gtk-3.0, see Fig. 14 for an example.

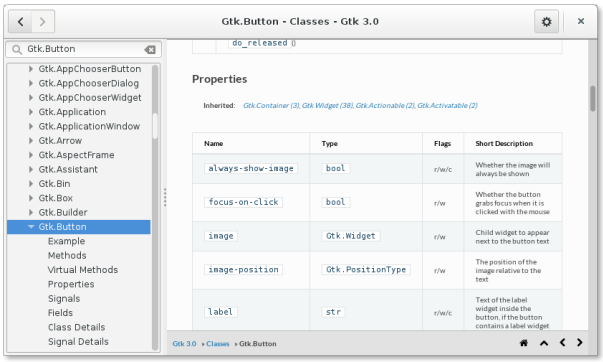


Fig. 15 The Devhelp documentation browser showing PGI-Docgen generated pages and the search index.

Furthermore the documentation provides the following features:

Devhelp integration. Devhelp, the official documentation browser of the Gnome project provides a combined view of the documentation of all Gnome projects. Making it possible to include the Python documentation reduces the number of user interfaces a developer has to deal with. See Fig. 15 for how the Devhelp integrations looks like.

Online and offline version. Online documentation makes it easy to look up documentation in cases were installation is not an option. Offline documentation is needed for cases where an Internet connection is not available or where low latency access is preferred.

Local search. It provides fast and configurable symbol search, which works both in online and offline mode.

Source Code Mapping. It links Python functions to the location of the C source implementation, allowing developers to see the actual implementation in case the documentation is lacking.

Source Code Mapping

In theory the provided documentation contains everything the developer would need for understanding the behavior of a function. In reality the documentation can be lacking due to missing information on error handling or allowed value ranges. Another problem specific to GObject Introspection is that the function annotations might be missing or wrong and as a result the function exposes an unusable Python interface. To make it easy to look at the C implementation of each function PGI-Docgen adds a link to each function leading to the source code exposed in the respective online version control repository browser, such as https://git.gnome.org.

For this to work we need the to fetch the debug information for each library, which is provided by the AutomaticDebugPackages project in Debian. With "objdump" we can list all the exported symbols and their address:

```
>>> objdump -t -j .text libgtk-3.so | grep -n1 "gtk_button_new$"
11317-0000000000152b30 g     F .text        000000000000009a gtk_color_button_set_alpha
11318:000000000012cb00 g     F .text        0000000000000019 gtk_button_new
11319-00000000002be390 g     F .text        0000000000000274 gtk_selection_convert
```

Then iterating over the DWARF compile units we can find the nearest DW_TAG_subprogram to the exported symbol and the source file and line:

```
>>> objdump --dwarf=info libgtk-3.so | grep -B8 12cb00
 <2><32b6a5>: Abbrev Number: 0
 <1><32b6a6>: Abbrev Number: 52 (DW_TAG_subprogram)
    <32b6a7>   DW_AT_external    : 1
    <32b6a7>   DW_AT_name        : (indirect string, offset: 0x18c92): gtk_button_new
    <32b6ab>   DW_AT_decl_file   : 1
    <32b6ac>   DW_AT_decl_line   : 1235
    <32b6ae>   DW_AT_prototyped  : 1
    <32b6ae>   DW_AT_type        : <0x326cfe>
    <32b6b2>   DW_AT_low_pc      : 0x12cb00
    <32b6ba>   DW_AT_high_pc     : 0x19
```

Wrapping this process in a Python library we can look up the source location of symbols:

```
>>> debug.get_line_numbers_for_name("libgtk-3.so")["gtk_button_new"]
'gtk/gtkbutton.c:1235'
```

Fig. 16 The implementation source of "gtk_button_new", which can be reached by following the source link in the Python API documentation.

Combining the path information with the library version and a template for an online git repository browser we can build an online source URL (see Fig. 16):

```
>>> path = 'gtk/gtkbutton.c:1235'
>>> Project.for_namespace("Gtk").get_source_func("Gtk")(path)
'https://git.gnome.org/browse/gtk+/tree/gtk/gtkbutton.c?h=3.22.7#n1235'
```

Conclusion and Future Work

This paper presented a range of benchmarks comparing different ways to access shared libraries in Python with both CPython and PyPy and showed that performance wise PyPy &

cffi can compete with C extensions written for CPython even if value validation and error handling is implemented in Python.

Based on these insights, a PyGObject compatible library, PGI, was implemented and presented which runs under both CPython and PyPy using the ctypes and cffi modules. In its current state PGI is only able to run smaller example programs and no real world applications. Further more PGI-Docgen was presented which generates complete and sound documentation based on PGI and GObject Introspection which is also helpful to existing users of PyGObject.

During the work on PGI and PGI-Docgen it became clear that PyGObject is lacking maintainership for the implementation and the surrounding documentation. Thus any work done in that area should be focused on PyGObject itself and not on PGI or other alternatives, since the prevalence of PGI also depends on the prevalence of PyGObject. The author of this paper has thus started to contribute to PyGObject instead of pursuing further improvements of PGI for the time being.

External Resources

These resources where created as part of this project and contain more information about their respective application or library:

The bechmarking code used for the presented results and plots:
https://bitbucket.org/lazka/pypy-gi-bench

The GIT repository of the PGI library:
https://github.com/lazka/pgi

The GIT repository of the PGI-Docgen program:
https://github.com/lazka/pgi-docgen

The online version of the documentation generated with PGI-Docgen:
https://lazka.github.io/pgi-docs/

The GIT repository of PGI-Docgen generated Devhelp packages:
https://github.com/pygobject/pgi-docs-devhelp

The PYPUI example framework:
https://github.com/lazka/pypui