Reaching Python from Racket

Pedro Palma Ramos
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
pedropramos@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Racket is a descendant of Scheme, a language that has been widely used to teach computer science. Recently, the Python language has taken over this role, mainly due to its huge standard library and the great number of third-party libraries available. Given that the development of equivalent libraries for Racket is an enormous task that cannot be currently done in an acceptable time frame, the next best option is to allow the Racket platform to use Python programs and libraries.

We have been developing an implementation of Python for the Racket platform based on a source-to-source compiler. In order to provide good performance and good interoperability with the Racket platform, the runtime libraries are being implemented over Racket data-types. This, however, entails implementing all of Python's standard library on Racket and it does not provide access to popular Python libraries implemented using C module extensions (such as Numpy and SciPy).

This paper presents an alternative approach that allows libraries from Python's reference implementation to be imported and used in our Racket implementation of Python, immediately providing access to all of Python's standard library and every third-party library, including NumPy and SciPy.

The proposed solution involves importing Python module objects directly from Python's virtual machine, by calling the Python/C API through Racket's Foreign Function Interface, and converting them to objects usable by our Racket runtime libraries, making them compatible with the Racket platform.

This compatibility layer therefore relies on relatively expensive foreign function calls to Python's libraries, but our performance tests show that the overhead introduced by them is quite low and, for most cases, it can be minimized in order to attain the same performance as Python's reference implementation.

ILC '14, August 14 - 17 2014, Montreal, QC, Canada
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2931-6/14/08$15.00.
http://dx.doi.org/10.1145/2635648.2635660

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors

General Terms
Languages

Keywords
Python; Racket; Interoperability

1. INTRODUCTION

The Racket programming language, a descendant of Scheme, has been used for introductory programming courses, particularly due to its focus on pedagogy. The Racket platform is shipped with DrRacket [4], an integrated development environment (IDE) specially tailored for inexperienced programmers, as it provides a simple and straightforward graphical interface. Furthermore, Racket and DrRacket support the development of additional programming languages [14], which allow users to write programs with modules in multiple programming languages.

More recently, the Python programming language has been replacing Scheme and Racket in many computer science courses. Among them, there is the rather infamous case of MIT dropping its introductory programming course based on Scheme (6.001 - Structure and Interpretation of Computer Programs) and replacing it with a course based on Python for robotics applications [9].

Python is a high-level, dynamically typed programming language [16, p. 3]. It supports the functional, imperative and object-oriented programming paradigms and features automatic memory management. Python's focus on readability and its huge standard library have made it a very popular language in many areas, which has led to a great variety of libraries being developed for Python. Furthermore, Python's reference implementation, CPython, allows developers to write module extensions in the C programming language, for high-performance libraries. This has led to some interesting libraries, namely NumPy [10], a library for high-performance array access widely used by the scientific computing community.

Being able to take advantage of Python and its libraries would be very valuable to the Racket platform. This was one of our main motivations for developing a Racket implementation of Python that allows interoperability between Python and Racket [11].

So far, we have explored two alternative strategies for implementing Python's runtime. The first one relied on using a foreign function interface [1] to map Python's operations into foreign calls to the Python/C API [18]. This allowed us to access every library supported by CPython, but, on the other hand, it suffered from two problems: (1) simple operations need to perform a significant number of foreign calls, which led to an unacceptably slow performance and (2) Python values would have to be explicitly converted to their Racket representation when mixing Python and Racket code, resulting in a clumsy interoperability.

Our second and current strategy consists of reimplementing Python's semantics and built-in data-types in Racket. This led to huge performance gains, even surpassing CPython's performance for certain programs. Also, since most Python data-types map directly to the corresponding ones in Racket, interoperability between both languages feels much more natural. On the other hand, this strategy entails reimplementing all of Python's standard library and it does not provide us with access to Python libraries based on C module extensions (such as NumPy).

In this paper, we present a mechanism for importing and converting Python libraries from CPython's virtual machine to our current Racket-based data model. This way, we attempt to get the best of both worlds, by keeping the enhanced performance and native Racket-Python interoperability obtained from reimplementing Python's runtime behaviour in Racket, while still being able to universally access every library available for CPython.

2. RELATED WORK

Before we present our own solution, this section will briefly describe other attempts to access CPython's libraries on alternative Python implementations.

2.1 Ironclad

Ironclad is an open-source project developed by William Reade and supported by Resolver Systems [7], whose goal is to make Python C module extensions available to IronPython (an implementation of Python for Microsoft's .NET platform [6]), most notably NumPy and SciPy.

This goal is actually very similar to ours. While Ironclad tries to bring together a C and a C# implementations with potentially very different internals, we are trying to bring together a C and a Racket implementations which are also potentially very different.

Ironclad tries to achieve this by replacing the library implementing the Python/C API with a stub which intercepts Python/C API calls and impersonates them using IronPython objects instead of the usual CPython objects.

For objects whose types are defined in a compiled C module extension, they have an IronPython type which wraps around them and forwards all method calls to the real Python/C API.

NumPy and SciPy already work with Ironclad. No benchmarks are provided, however the author mentions that performance is generally poor compared to CPython. He claims that ”in many places it's only a matter of a few errant microseconds (...) but in pathological cases it's worse by many orders of magnitude” [3].

2.2 JyNI

JyNI is another compatibility layer, being developed by Stefan Richthofer [8], whose goal is similar to Ironclad's but it's meant for Jython instead of IronPython. Jython is an alternative Python implementation, written in Java, and designed to be able to interact with the Java Virtual Machine.

It is still in an early phase of development (alpha) and does not yet support NumPy, but it already supports some of Python's built-in types.

It uses a mix of three strategies for bridging objects from CPython to Jython and vice-versa [12]:

1. Like Ironclad, it loads a stub of the Python/C API library which delegates its calls to Jython objects. This only works for types which are known to Jython and where the Python/C API uses no preprocessor macros to directly access an object's memory (because the stub would not know how to map these pointer offsets);

2. For the types where the Python/C API uses preprocessor macros, objects created on the CPython side are mirrored on the Jython side. For immutable objects this is trivial because there is no need for further synchronization. Mutable objects are mirrored with Java interfaces which provide access to the object's shared memory;

3. Finally, types unknown to Jython (because they are defined in a C module extension) or opaque types are wrapped by a Jython object which forwards method calls to the Python/C API and converts arguments and return values between their CPython and Jython representations.

2.3 CLPython

CLPython is another alternative Python implementation, this one written in Common Lisp. It was being developed by Willem Broekema since 2006, but the project has been officially halted since 2013. Its goal was to bridge Python and Common Lisp development, as it allows accessing Python libraries from Common Lisp and vice-versa, as well as mixing Python and Common Lisp code.

CLPython was one of the sources of inspiration for our second strategy to the runtime implementation, as it also maps Python objects to equivalent Common Lisp values whenever possible. Unfortunately, it does not provide support for C module extensions, since it does not implement the Python/C API [2].

3. SOLUTION

In this section we will describe the architecture and the major decisions we took for bridging CPython with the Racket platform. In order to agree on a common terminology and make the trade-offs of our decisions clear, we will start by briefly going over Python's data model and how it is represented on our Racket runtime implementation.

3.1 Python's Data Model

In Python, every value is treated as an instance of an object, including basic types such as integers. Every object has a reference to its type, which is represented by a type-object (also a Python object). A type-object contains a tuple with its supertypes and a dict (or dictionary, Python's name for a hash-table) which maps attribute and method names to the attributes and methods themselves.

The way an object behaves to each of the language's operators is stored in its type-object's dict, as a method. For instance, the expression `a + b` (adding objects `a` and `b`) is equivalent to `type(a).__add__(a,b)`. This technique is used for all unary and binary operators, for getting/setting an attribute/index/slice, for printing objects, for obtaining their length, etc [17].

For user-defined types, these methods can be defined during class creation (a `class` statement defines a new type-object), but they may also be changed dynamically at runtime, by adding, updating or removing these entries from the type-objects' dictionary, through reflection.

A Python source file can be imported into a module, which is also a Python object and a first-class citizen. A Python module contains a hash-table which stores the variables, functions and classes which are defined in that file. As with type-objects, these values can be accessed as if they were fields or methods of the module.

For our implementation, we tried to map Python's built-in data-types directly onto Racket data-types whenever possible. To name some:

- Python's numerical tower (`int`, `long`, `float`, `complex`) is mapped to Racket numbers;
- Python's Boolean values (`True` and `False`) are a subtype of `int`, but they are mapped to Racket's Boolean values (`#t` and `#f`) and converted to the integers 1 and 0 when needed;
- Python's strings are directly mapped to Racket strings;
- Python's dicts are directly mapped to Racket hash-maps;
- Python's tuples are immutable and have O(1) access time, so they are mapped to Racket immutable vectors.

Other data types are mapped to structures whose first field is a reference to their type object. For instance, Python's lists are mutable and also have O(1) access time, so they are mapped to a structure containing a vector, so that operations which alter the list's size can allocate a new vector, without affecting the object's identity.

As mentioned before, most Python operations require computing an object's type in order to lookup a method in its hash-table. Since the data-types which are directly mapped to Racket data-types do not store a reference to their type-objects, we compute them through a pattern matching function which returns the most appropriate type-object, according to the predicates satisfied by the value.

3.2 Importing Modules from CPython

There are 3 possible syntaxes for importing a module's bindings in Python:

- `import <module>` - makes <module> available as a new binding for a module object. The bindings defined inside that module are accessible as attributes and methods;
- `from <module> import <id>` - the <id> binding defined inside <module> is made available as a new binding. No binding is provided for the module object;
- `from <module> import *` - similar to the above, but provides all bindings defined inside <module>.

We have mapped these import statements to uses of Racket's `require` and `dynamic-require` forms. This works because the imported Python modules are first compiled to Racket.

Importing a module directly from CPython requires a radically different approach, therefore, we now provide a slight change to the syntax described above for explicitly importing modules from CPython: replacing "`import`" with "`cpy-import`".

The module objects themselves are imported from the Python/C API using the Racket Foreign Function Interface (FFI), which returns a C pointer to the module object allocated by CPython in shared memory. In order to make it compatible with our runtime operations, we convert this foreign object to our Racket representation of a module. This entails recursively converting the contents of that module's hash-table to their Racket representations.

We achieve this by defining these two general-purpose functions:

- `cpy->racket`, takes a foreign object C pointer as input and builds its corresponding value according to our Racket representation;
- `racket->cpy`, takes a Racket value as input and returns a C pointer to its corresponding Python object allocated in CPython.

Both functions start by figuring out the argument's type and then dispatch its conversion to a more specific function. An excerpt of their implementations is presented below.

```
(define (cpy->racket x)
  (let ([type (PyString_AsString
                (PyObject_GetAttrString
                 (PyObject_Type x) "__name__"))])
    (case type
      [("bool") (bool-from-cpy x)]
      [("int") (int-from-cpy x)]
      ...
      [else (proxy-obj-from-cpy x)])))
```

```
(define (racket->cpy x)
  (cond
    [(boolean? x) (make-cpy-bool x)]
    ...
    [(proxy-object? x) (unwrap-proxy-object x)]
    [else (error "racket->cpy:␣not␣supported:" x)]))
```

The following sections describe in detail how the different types are converted.

3.3 Converting Basic Types

Basic immutable types (`bool`, `int`, `float`, `complex` and `string`) are trivially converted to their Racket representations (the Python/C API provides functions for these conversions).

Since these objects are immutable, their identity is not relevant, therefore there is no need to keep track of the original C pointers or to synchronize potential changes between the Racket and CPython virtual machines.

3.4 Converting Type-Objects

Like with modules, it is essential that we convert type-objects to a representation that is compatible with our runtime operations, especially because most Python operations rely on fetching attributes from these type-objects.

The structure of a type-object is constant and straightforward, even when that type was defined in the library we are importing. Among other less important attributes, a type-object contains a name, a tuple with its supertypes and a hash-table containing its fields and methods.

Again, as with module objects, we convert imported type-objects by building our own type-object according to Racket's representation, i.e., recursively converting the type-objects that make up its supertypes tuple and the entries that make up its hash-table.

We also keep a global hash-map as a cache for imported type-objects. It maps a C pointer to its converted type-object. This way, before attempting to convert a new type-object from CPython, we first check the cache to see if its C pointer is here, and if so, a reference to the already converted type-object is returned.

3.5 Converting Opaque Objects

The default case when converting an object from CPython is implemented by the `proxy-obj-from-cpy` function. This one simply wraps the C pointer and its converted type-object in a Racket structure that we call a *proxy object*.

Like its name suggests, a proxy object acts as a proxy, in Racket, for the Python object in CPython's shared memory. It is especially suited for objects whose internal representation we do not know, such as the types defined in the libraries we are importing, but we also use them to wrap around other opaque objects (e.g. Python functions) and mutable objects which could be updated "behind our backs" (e.g., lists and dicts).

Converting a proxy object back to its CPython representation is as easy as unwrapping its C pointer.

In order for proxy objects to be applied as Racket procedures, we take advantage of the fact that structures are applicable in Racket. To this end, we define the following `prop:procedure` structure property [5]:

```
1  (lambda (f . args)
2    (let ([ffi_call_result
3           (PyObject_CallObject
4            (unwrap-proxy-object f)
5            (list->cpy-tuple (map racket->cpy args)))])
6      (if ffi_call_result
7          (cpy->racket ffi_call_result)
8          (let ([cpy-exception (second (PyErr_Fetch))])
9            (raise (cpy->racket cpy-exception))))))
```

In order for the object to be called (line 3), the C pointer inside the proxy object is unwrapped (line 4) and the arguments are converted to their CPython representations and packed into a tuple (line 5).

This will return a C pointer to a Python object if the call is successful or return `#f` (false) if it resulted in an unhandled exception. In the former case, the call result is converted to Racket and returned (line 7), while in the latter one, the exception is fetched, converted to Racket and re-raised (lines 8-9).

Notice that this strategy allows our implementation to transparently handle any Python operation on imported objects of any type known to CPython. When a type object is imported and converted, its methods are converted and stored as proxy objects.

For instance, adding two proxy-objects entails fetching the `__add__` method from the type-object (proxy-objects store a reference to their converted type-objects) and calling it with the two proxy-objects as arguments. This method call is handled by CPython (via FFI) and its result is then converted or wrapped in a proxy object, closing the cycle.

The semantics on exception handling is also well integrated with our implementation. Since we keep a cache for the converted type-objects, we assure that each type-object from CPython is only converted once and all references to that type will point to same object. Therefore, when determining if an exception handling clause should handle an exception, type-objects can be safely compared by `eq?`.

3.6 Dealing with Heterogeneity

As mentioned earlier, we convert collections (lists, tuples, sets and dicts) by wrapping them as proxy objects. One of the reasons for this is that in a scenario where the user would need to pass a huge collection as argument or return value of a proxy object call, converting such collection back and forth would be a big bottleneck.

The strongest reason, though, is that copying a mutable collection's contents to a new collection would not respect the object's identity and would lead to implementing the wrong semantics. Consider the example of importing the following module, where a function logs its arguments to a globally defined list.

```
log = []

def foo(n):
  log.append(n)
  return n * n
```

This module provides the `log` list and the `foo` function. It should be clear that the `foo` function should be imported as a proxy object, so that its calls are handled by CPython's virtual machine.

Suppose that we import and convert `log` to our list representation. When we call the `foo` proxy object, it updates the `log` list in CPython's shared memory, and not our converted list. The only way to keep track of the changes to `log` is by accessing its original C pointer via a proxy object.

This solution, however, leads to another issue: we now have objects of the same type with two distinct and heterogeneous representations. In the case of lists, we have the standard representation as a boxed vector and the proxy object representation. Even though both of them implement Python's semantics correctly when used independently, proxy object lists cannot be transparently used for operations with standard lists.

We try to correct this by giving the user the power to explicitly convert proxy object collections to their standard representation. Python type names generally act as constructors for converting or copying objects from other types to that type. For instance, consider a tuple `a` and a list `b`. The expression `list(a)` returns a new list with the contents of tuple `a`, while `list(b)` effectively returns a shallow copy of list `b`.

We can overload the `list`, `tuple`, `set` and `dict` constructors for proxy object collections to act as explicit converters to their standard representations. This is in accordance with the original semantics of these constructors because it acts both as a type conversion and a shallow copying mechanism.

3.7 Using Python Libraries in Racket

So far we have described how these imported libraries interoperate with our Python implementation, however this mechanism can also act as a standalone library for Racket, simply by requiring this module:

```
> (require "cpy-importing.rkt")
```

The `cpy-import...` and `from...cpy-import...` syntaxes are implemented by the `cpy-import` and `cpy-from` macros, respectively. Let us import `date` from Python's `datetime` module [15].

```
> (cpy-from "datetime" import (["date" as date]))
```

The identifier `date` is now available as a type-object. Let us get today's date (imagining that today is 14 August 2014, International Lisp Conference's first day). This is done by getting the `today` function from the `date` type-object and calling it without arguments.

```
> (define ilc ((py-get-attr date "today")))
```

The obtained value is a proxy-object, but we can print it using Python's string representation.

```
> ilc
(proxy-object ... #<cpointer:PyObject>)
> (py-print ilc)
2014-08-14
```

We can also get its attributes and call its methods. Notice that Python integers are seamlessly converted to Racket integers.

```
> (define ilc-year (py-get-attr ilc "year"))
> ilc-year
2014
> (integer? ilc-year)
#t
> (define ilc-weekday (py-method-call ilc "isoweekday"))
> ilc-weekday
4
> (integer? ilc-weekday)
#t
```

Let us count how many days are left until Christmas by subtracting both dates. Python's minus operator is available as the function `py-sub`.

```
> (define christmas (date 2014 12 25))
> (define interval (py-sub christmas ilc))
> (py-get-attr interval "days")
133
```

As mentioned previously, collections are converted to proxy-objects by default, so the Python tuple returned below cannot be directly manipulated in Racket.

```
> (py-method-call ilc "isocalendar")
(proxy-object ... #<cpointer:PyObject>)
```

However, we do provide the same functionality we use for the `list`, `tuple`, `set` and `dict` constructors to convert them to Racket representations. Let us convert this Python tuple to a Racket vector.

```
> (define iso-calendar
    (tuple-from-cpy
     (unwrap-proxy-object
      (py-method-call ilc "isocalendar"))))
> (vector? iso-calendar)
#t
> iso-calendar
'#(2014 33 4)
> (vector-ref iso-calendar 0)
2014
```

The bindings for the remaining collections are similar. Python lists, for instance, would be converted with `list-from-cpy`, and since they are implemented as a structure wrapping a vector, we further provide the bindings `py-list->vector` and `py-list->list` for conveniently converting them to Racket vectors or lists.

4. PERFORMANCE EVALUATION

In this section, we present some benchmarks for measuring the overhead introduced by our type conversions when using imported libraries from CPython.

These benchmarks were performed on an Intel® Core^{TM} i7 processor at 3.2GHz running under Windows 7. The measured times represent the minimum out of 5 samples.

Consider the Python example below, using the NumPy library, where we define and call a function which adds a given number of 100×100 matrices with random integers up to 100000.

```
import numpy as np

def add_arrays(n):
  result = np.zeros((100,100))
  for i in range(n):
    result += np.random.randint(0, 100000, (100,100))
  return result

print add_arrays(10000)
```

To get this code running on the Racket platform, we simply have to declare its language with `#lang python` as the first line and replace `import` with `cpy-import`.

Using CPython, we get a minimum running time of 1890ms, while using the Racket platform, we get a minimum running time of 2464ms (about 30% slower).

While such an overhead is very acceptable for most use cases, we realize that it may be an issue for high-performance applications. Fortunately, it is possible to come up with an alternative which virtually eliminates the overhead from FFI calls and type conversions.

We can define the `add_arrays` function as a CPython library (`arrays_example.py`, in this case) and import it and call it on the Racket platform, like this:

```
#lang python
from arrays_example cpy-import add_arrays

print add_arrays(10000)
```

This way, instead of dealing with FFI calls and type conversions on every iteration of the `for` cycle, we simply have one foreign function call to deal with, since the computation of `add_arrays` is handled by CPython. The minimum running time measured for this example was now 1887ms, which is identical to the one measured for CPython.

This is an impressive figure when compared to what can be achieved with traditional Racket libraries. Using Typed Racket, a statically typed dialect of Racket which provides enhanced performance [13], and `math/matrix`, a Racket library for processing matrices, an equivalent program would be written as:

```
#lang typed/racket
(require math/matrix)

(: randint (Integer Integer Integer
            -> (Matrix Integer)))
(define (randint x y limit)
  (build-matrix x y (lambda (x y) (random limit))))

(: add-arrays (Integer -> (Matrix Integer)))
(define (add-arrays n)
  (let: ([result : (Matrix Integer)
                 (make-matrix 100 100 0)])
    (for ([i n])
      (set! result
            (matrix+ result
                     (randint 100 100 100000))))
    result))

(add-arrays 10000)
```

For this program (running on the Racket platform), we get a minimum running time of 25568ms, about 13 times slower than what is achieved with NumPy, using our import mechanism.

5. CONCLUSIONS

Our previous efforts to implement the Python language for the Racket platform had resulted in:

(a) a runtime module based on CPython's object representation which had universal access to all Python libraries available to CPython, but suffered from very poor performance due to the bottleneck caused by the use (and abuse) of Racket's Foreign Function Interface and also did not allow for a native interoperability between other Racket languages since types had to be explicitly converted;

(b) a runtime module based on Racket's value representation, which had a very acceptable speed and native interoperability with other Racket languages, but needed a reimplementation of all of Python's standard library and could not access libraries based on C module extensions.

We have now developed a complementary import system which imports modules from CPython's virtual machine using the Python/C API and Racket's FFI and recursively converts each imported data-type to a Racket representation compatible with the runtime module from (b).

With this additional import system, we are now able to access any library installed on CPython while keeping a good integration with other languages for the Racket platform, since basic data-types are automatically converted to Racket when imported, and keeping an acceptable performance for the remainder of the Python language, since our runtime module remains unchanged.

Furthermore, it is possible to minimize the number of FFI calls and type conversions on computationally intensive blocks of code by moving their computation to CPython and expressing them as function calls, therefore obtaining the same performance as in CPython.

Finally, it is worth mentioning that while these features were designed to be used for a Python runtime implementation, it is possible to use them as a stand-alone Racket library which acts like an API for accessing Python libraries on Racket programs.

6. ACKNOWLEDGMENTS

This work was partially supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia under contract Pest-OE/EEI/LA0021/2013 and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

7. REFERENCES

[1] E. Barzilay. *The Racket Foreign Interface*, 2012.

[2] W. Brokema. *CLPython Manual*, chapter 10.6 Compatibility with CPython C extensions. 2011.

[3] J. D. Cook. Numerical computing in IronPython with Ironclad. `http://www.johndcook.com/blog/2009/03/19/ironclad-ironpytho/`. [Online; retrieved on May 2014].

[4] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

[5] M. Flatt. *The Racket Reference*, chapter 4.17 Procedures. 2013.

[6] J. Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004 International Python Conference*, volume 8, 2004.

[7] Ironclad - Resolver Systems. `http://www.resolversystems.com/products/ironclad/`. [Online; retrieved on May 2014].

[8] JyNI – Jython native interface. `http://www.jyni.org/`. [Online; retrieved on May 2014].

[9] L. Kaelbling, J. White, H. Abelson, D. Freeman, T. Lozano-Pérez, and I. Chuang. 6.01sc introduction to electrical engineering and computer science i, spring 2011. `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011`. [Online; retrieved on June 2014].

[10] NumPy. `http://www.numpy.org/`. [Online; retrieved on May 2014].

[11] P. P. Ramos and A. M. Leitão. An implementation of Python for Racket. In *7th European Lisp Symposium*, page 72, 2014.

[12] S. Richthofer. JyNI - using native CPython-extensions in Jython. In *EuroSciPi 2013*, Brussels, Belgium, 2013.

[13] S. Tobin-Hochstadt and V. St-Amour. *The Typed Racket Guide*, 2013.

[14] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. *ACM SIGPLAN Notices*, 46(6):132–141, 2011.

[15] G. van Rossum and F. L. Drake. *Python library reference*, chapter 8.1 datetime - Basic date and time types. Centrum voor Wiskunde en Informatica, 1995.

[16] G. van Rossum and F. L. Drake. *An introduction to Python*. Network Theory Ltd., 2003.

[17] G. van Rossum and F. L. Drake. *The Python Language Reference*, chapter 3. Data model. 2010.

[18] G. Van Rossum and F. L. Drake Jr. *Python/C API reference manual*, 2002.