Implementing Python for DrRacket

Pedro Palma Ramos and António Menezes Leitão

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, Lisboa, Portugal
`{pedropramos,antonio.menezes.leitao}@tecnico.ulisboa.pt`

Abstract

The Python programming language is becoming increasingly popular in a variety of areas, most notably among novice programmers. On the other hand, Racket and other Scheme dialects are considered excellent vehicles for introducing Computer Science concepts. This paper presents an implementation of Python for Racket and the DrRacket IDE. This allows Python programmers to use Racket libraries and vice versa, as well as using DrRacket's pedagogic features. In particular, it allows architects and designers to use Python as a front-end programming language for Rosetta, an IDE for computer-aided design, whose modelling primitives are defined in Racket.

Our proposed solution involves compiling Python code into equivalent Racket source code. For the runtime implementation, we present two different strategies: (1) using a foreign function interface to borrow the data types and primitives from Python's virtual machine or (2) implementing Python's data model over Racket data types.

While the first strategy is easily implemented and provides immediate support for Python's standard library and existing third-party libraries, it suffers from performance issues: it runs, at least, one order of magnitude slower when compared to Python's reference implementation.

The second strategy requires us to implement Python's data model in Racket and port all of Python's standard library, but it succeeds in solving the former's performance issues. Furthermore, it makes interoperability between Python and Racket code easier to implement and simpler to use.

1998 ACM Subject Classification D.3.4 Programming Languages: Processors

Keywords and phrases Python, Racket, language implementations, compilers

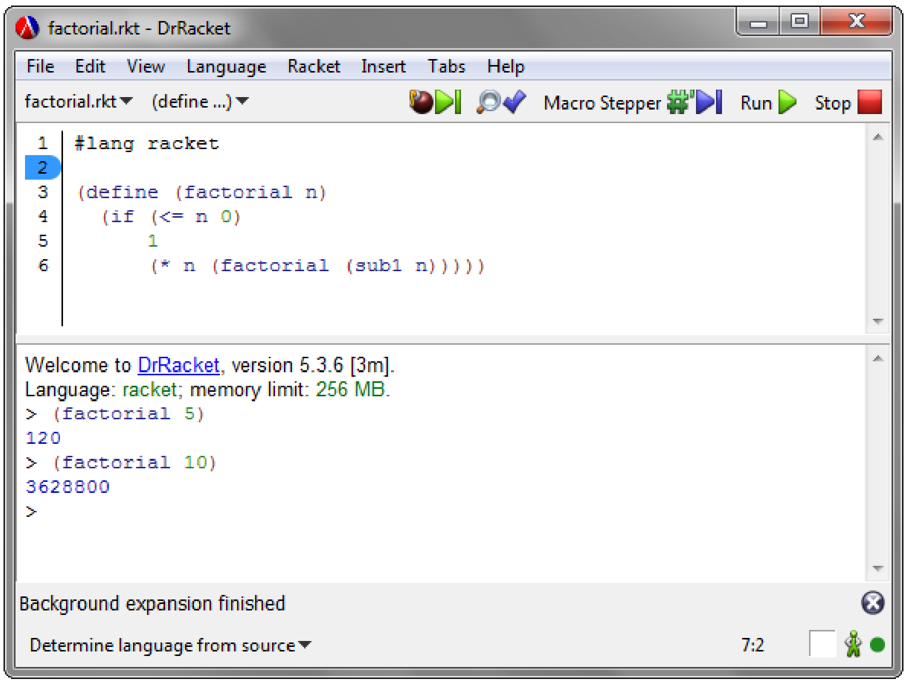
Digital Object Identifier 10.4230/OASIcs.SLATE.2014.127

1 Introduction

Architects who use computer-aided design (CAD) applications are beginning to shift from a traditional approach to an algorithmic approach. This is leading to an increasing need for the CAD community to master generative design, a design method based on a programming approach which allows them to build complex three-dimensional structures that can then be effortlessly modified through simple changes in a program's code or parameters. It is, thus, increasingly important for architects to master programming techniques.

Although most CAD applications provide programming languages for generative design, programs written in these languages have very limited portability, as each CAD application provides its own specific language and functionality. Therefore, a program written for one CAD application cannot be used on other CAD applications. In addition to this, these programming languages are rarely pedagogical and most of them are poorly designed or obsolete.

With this in mind, we are developing Rosetta, an extensible integrated development environment (IDE) for generative design [11]. Rosetta seeks to answer the portability problem by allowing the development of programs in different programming languages


factorial.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
factorial.rkt (define ...)
Macro Stepper
Run
Stop

```
#lang racket

(define (factorial n)
  (if (<= n 0)
      1
      (* n (factorial (sub1 n)))))
```

```
Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 256 MB.
> (factorial 5)
120
> (factorial 10)
3628800
>
```

Background expansion finished

Determine language from source 7:2

Figure 1 DrRacket's graphical user interface.

which are then portable across different CAD applications. A program written in one of the supported programming languages is compiled to an intermediate language, where the primitives essential to 3D modelling are defined. These primitives are then translated to commands of the selected CAD application's interface and invoked through interprocess communication. This design allows Rosetta to support new front-end programming languages and new back-end CAD applications, independently from each other.

Currently, Rosetta is based on the DrRacket IDE and uses Racket as its intermediate language. DrRacket (formerly known as DrScheme) is a pedagogic IDE for the Racket programming language, a dialect of LISP and a descendant of Scheme [3, 4].

Unlike IDEs such as Eclipse or Microsoft Visual Studio, DrRacket provides a simple and straightforward interface (Figure 1). It also provides a set of tools, including a syntax highlighter, a syntax checker, a macro stepper and a debugger, aimed at inexperienced programmers, such as the target audience of Rosetta.

Additionally, Racket and DrRacket support the development and extension of other programming languages [20], which is crucial for Rosetta's front-end extensibility. Currently, Rosetta supports front-ends for Racket, AutoLISP, JavaScript and RosettaFlow (a graphical language inspired by Grasshopper). AutoLISP and JavaScript were chosen precisely because they have been used for generative design. More recently, the Python language has emerged as a good candidate for this area of application.

Python is a high-level, interpreted, dynamically typed programming language [23, p. 3]. It supports the functional, imperative and object-oriented programming paradigms and features automatic memory management. It is mostly used for scripting, but it can also be used to build large scale applications. Its reference implementation, CPython, is written in C and it is maintained by the Python Software Foundation. There are also other third-party implementations such as Jython (written in Java), IronPython (written in C#) and PyPy (written in Python).

Due to its large standard library, expressive syntax and focus on code readability, Python is becoming an increasingly popular programming language in many areas, including architecture. Python has been receiving a lot of attention in the CAD community, particularly after it has been made available as scripting language for CAD applications such as Rhino or Blender. This justifies the need for implementing Python as another front-end language of Rosetta, i.e. implementing Python in Racket.

Therefore, our goal is to develop a correct and efficient implementation of the Python language for Racket, which is capable of interfacing Python and Racket code. This will allow Rosetta users to use Python as a front-end programming language for generative design. Additionally, we want to support some of DrRacket's features for Python development, namely syntax highlighting, syntax checking and debugging.

In the next sections, we will briefly examine the strengths and weaknesses of other Python implementations, describe the approaches we took for our own implementation and showcase the results we have obtained so far.

2 Related Work

There are a number of Python implementations that are good sources of ideas for our own implementation. In this section we describe the most relevant ones.

2.1 CPython

CPython, maintained by the Python Software Foundation, is written in the C programming language and has been the reference implementation of Python since its first release. It parses Python source code (from `.py` files or interactive mode) and compiles it to bytecode, which is then interpreted on a virtual machine.

The Python standard library is implemented both in Python and C. In fact, CPython makes it easy to write third-party module extension in C to be used in Python code. The inverse is also possible: one can embed Python functionality in C code, using the Python/C API [22].

2.1.1 Object Representation

CPython's virtual machine is a simple stack machine, where the byte codes operate on a stack of `PyObject` pointers [21].

At runtime, every Python object has a corresponding `PyObject` instance. A `PyObject` contains a reference counter, used for garbage collecting, and a pointer to a `PyTypeObject`, which is another `PyObject` that indicates the object's type. In order for every value to be treated as a `PyObject`, each built-in type is declared as a structure containing these two fields, plus any additional fields specific to that type.

This means that everything is allocated on the heap, even basic types. To avoid relying too much on expensive dynamic memory allocation, CPython enforces two strategies:

- Only requests larger than 256 bytes are handled by `malloc` (the C standard allocator), while smaller ones are handled by pre-allocated memory pools.
- There is a pool for commonly used immutable objects (such as the integers from -5 to 256). These are allocated only once, when the virtual machine is initialized. Each new reference to one of these integers will point to the instance on the pool instead of allocating a new one.

2.1.2 Garbage Collection and Threading

Garbage collection in CPython is performed through reference counting. Whenever a new Python object is allocated or whenever a new reference to it is made, its reference counter is incremented. When a reference is no longer needed, the reference counter is decremented. When the reference counter reaches zero, the object's finalizer is called and the space is reclaimed.

Reference counting, however, does not work well with reference cycles [24, ch. 3.1]. Consider the example of a list containing a reference to itself. When its last reference goes out of scope, its counter is decremented, however the circular reference inside the list is still present, so the reference counter will never reach zero and the list will not be garbage collected, even though it is already unreachable.

Furthermore, these reference counters are not thread-safe [25]. If two threads would attempt to increment an object's reference counter simultaneously, it would be possible for this counter to be erroneously incremented only once. To avoid this from happening, CPython enforces a global interpreter lock (GIL), which prevents more than one thread running interpreted code at the same time.

This is a severe limitation to the performance of threads on CPU-intensive tasks. In fact, using threads will often yield a worse performance than using a sequential approach, even on a multiple processor environment [1]. Therefore, the use of threads is only recommended for I/O tasks [2, p. 444].

Note that the GIL is a feature of CPython and not of the Python language. This feature is not present in other implementations such as Jython or IronPython, which will be described in the following section.

2.2 Jython

Jython is another Python implementation, written in Java and first released in 2000. Similarly to how CPython compiles Python source-code to bytecode that can be run on its virtual machine, Jython compiles Python source-code to Java bytecode, which can then be run on the Java Virtual Machine (JVM).

2.2.1 Implementation Differences

There are some aspects of Python's semantics in which Jython's implementation differs from CPython's [10]. Some of these are due to limitations imposed by the JVM, while others are considered bugs in CPython and, thus, were implemented differently in Jython.

The standard library in Jython also suffers from minor differences from the one implemented in CPython, as some of the C-based modules have been rewritten in Java.

2.2.2 Java Integration

Jython programs cannot use module extensions written for CPython, but they can import Java classes, using the same syntax for importing Python modules.

There is work being done by a third-party [18] to integrate CPython module extensions with Jython, through the use of the Python/C API. This would allow using NumPy and SciPy with Jython, two very popular Python libraries which rely on CPython's ability to run module extensions written in C.

2.2.3 Performance

It is worth noting that garbage collection is performed by the JVM and does not suffer from the issues with reference cycles that plague CPython [9, p. 57]. Furthermore, there is no global interpreter lock, so threads can take advantage of multi-processor architectures for CPU-intensive tasks [9, p. 417].

Performance-wise, Jython claims to be approximately as fast as CPython. Some libraries are known to be slower because they are currently implemented in Python instead of Java (in CPython these are written in C). Jython's performance is also deeply tied to performance gains in the Java Virtual Machine.

2.3 IronPython

IronPython, developed as a follow-up to Jython, is an implementation of Python for the Common Language Infrastructure (CLI). It is written in C# and was first released in 2006. It compiles Python source-code to CLI bytecode, which can be run on Microsoft's .NET framework or Mono (an open-source alternative implementation of the CLI).

IronPython provides support for importing .NET libraries and using them with Python code [13]. As it happened with Jython, there is work being done by a third-party in order to integrate CPython module extensions with IronPython [8].

As far as performance goes, IronPython claims to be 1.8 times faster than CPython on `pystone`, a Python benchmark for showcasing Python's features. Additionally, further benchmarks demonstrate that IronPython is slower at allocating and garbage collecting objects and running code with `eval`. On the other hand, it is faster at setting global variables and calling functions [6].

2.4 PyPy

PyPy is yet another Python implementation, written in a restricted subset of Python, RPython[1]. It was first released in 2007 and currently its main focus is on speed, claiming to be 6.2 times faster than CPython in a geometric average of a comprehensive set of benchmarks [17].

It supports all of the core language, most of the standard library and even some third party libraries. Additionally, it features incomplete support for the Python/C API [16].

PyPy includes two very distinct modules: a Python interpreter and the RPython translation toolchain [15]. Like the implementations mentioned before, the interpreter converts user's Python source code into bytecode.

However, what distinguishes it from those other implementations is that this interpreter, written in RPython, is in turn compiled by the RPython translation toolchain, effectively converting Python code to a lower level platform (typically C, but the Java Virtual Machine and Common Language Infrastructure are also supported).

The translation toolchain consists of a pipeline of transformations (flow analysis, annotator, backend optimizations, among others), but what truly makes PyPy stand out as currently the fastest Python implementation is its just-in-time compiler (JIT), which detects common codepaths at runtime and compiles them to machine code, optimizing their speed.

[1] RPython (Restricted Python) is a heavily restricted subset of Python, in order to allow static inference of types. For instance, it does not allow altering the contents of a module, creating functions at runtime, nor having a variable holding incompatible types.

Table 1 Comparison between implementations.

	Language(s) written	**Platform(s) targetted**	**Speedup (vs CPython)**	**Std. library support**
CPython	C	CPython's VM	$1\times$	Full
Jython	Java	JVM	$\sim 1\times$	Most
IronPython	C#	CLI	$\sim 1.8\times$	Most
PyPy	RPython	C, JVM, CLI	$\sim 6\times$	Most
PLT Spy	Scheme, C	Scheme	$\sim 0.001\times$	Full

The JIT keeps a counter for every loop that is executed. When it exceeds a certain threshold, that codepath is recorded and compiled to machine code. This means that the JIT works better for programs without frequent changes in loop conditions.

2.5 PLT Spy

PLT Spy is an experimental Python implementation written in PLT Scheme and C, first released in 2003. It parses and compiles Python source-code into equivalent PLT Scheme code [12].

PLT Spy's runtime library is written in C and interfaces with Scheme via the PLT Scheme C API. It implements Python's built-in types and operations by mapping them to the CPython virtual machine, through the use of the Python/C API. This allows PLT Spy to support every library that CPython supports (including NumPy and SciPy).

This extended support has a big trade-off in portability, though, as it led to a strong dependence on the 2.3 version of the Python/C API library and does not seem to work out-of-the-box with newer versions. More importantly, the repetitive use of Python/C API calls and conversions between Python and Scheme types severely limited PLT Spy's performance. PLT Spy's authors use anecdotal evidence to claim that it is around three orders of magnitude slower than CPython.

2.6 Comparison

Table 1 displays a rough comparison between the implementations discussed above.

To sum up, PLT Spy can interface Python code with Scheme code and is the only alternative implementation which can effortlessly support all of CPython's standard library and third-party modules extensions, through its use of the Python/C API. However, the performance cost that results from the repeated conversion of data from Scheme's internal representation to CPython's internal representation is unacceptable.

Furthermore, our implementation will require using Racket's bytecode and tools in order to support Rosetta's modelling primitives (defined in Racket), therefore PyPy's performance strategy is not feasible for our problem.

On the other hand, Jython and IronPython show us that it is possible to implement Python's semantics over high-level languages, with very acceptable performances and still provide means for importing that language's functionality into Python programs. However, Python's standard library needs to be manually ported.

With this in mind, we will be presenting our proposed solution in the next section.

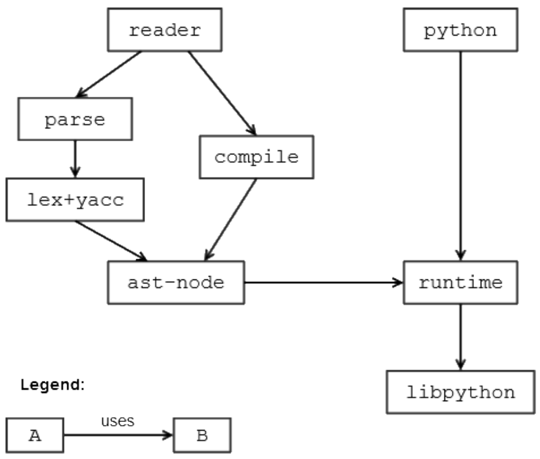


Figure 2 Dependencies between modules. The arrows indicate that a module uses functionality that is defined on the module it points to.

3 Solution

Our proposed solution consists of two compilation phases:

1. Python source-code is compiled to Racket source-code;
2. Racket source-code is compiled to Racket bytecode.

In phase 1, the Python source code is parsed into a list of abstract syntax trees, which are then expanded into a list of syntax objects containing equivalent Racket code.

In phase 2, the Racket source-code generated above is fed to a bytecode compiler which performs a series of optimizations (including constant propagation, constant folding, in-lining, and dead-code removal). This bytecode is interpreted on the Racket VM, where it may be further optimized by a JIT compiler.

Note that phase 2 is automatically performed by the Racket implementation, therefore our implementation effort relies only on a source-to-source compiler from Python to Racket.

3.1 General Architecture

Figure 2 summarises the dependencies between the different Racket modules of the proposed solution. The next paragraphs provide a more detailed explanation of these modules.

3.1.1 Racket Interfacing

A Racket file usually starts with the line `#lang <language>` to indicate which language is being used (in our case, it will be `#lang python`).

To define a language for the Racket platform, Racket requires only two files: one which defines how source code compiles to Racket code, and another which defines the functions and macros that make up the compiled code. These correspond, in our solution, to the `reader` and `python` modules, respectively.

The `python` module simply provides all the bindings from the Racket language and the bindings defined at the `runtime` module (which will be described at 3.1.3). The `reader` module must provide:

- The `read` function, which takes a Racket input-port as an argument (this could be the standard input, a file, a string, etc.) and return a list of s-expressions, which correspond to the Racket code compiled from the input port;
- The `read-syntax` function, which also takes an input-port as argument and returns a list of syntax-objects, which correspond to the compiled Racket code.

Syntax-objects are Racket's native representation for code. They abstract over s-expressions, but they may also keep source location information (file, line number, column number and span) and lexical binding information about said code. By keeping track of the original position of each token during the parsing process and copying it to the compiled syntax-objects, each compiled s-expression can be mapped back to the original Python expression it corresponds to. This way, our implementation fully integrates with DrRacket's features such as signalling the line number where an error has occurred or tracking the location of a debugging session on the source code.

3.1.2 Parse and Compile Modules

The `lex+yacc` module defines a set of Lex and Yacc rules for parsing Python code, using the Lex/Yacc implementation provided by Racket's `parser-tools` library. This outputs a list of abstract syntax trees (ASTs), which are defined in the `ast-node` module. These nodes are implemented as Racket objects. Each subclass of an AST node defines its own `to-racket` method, responsible for the code generation. A call to `to-racket` works in a top-down recursive manner, as each node will eventually call `to-racket` on its children.

The `parse` module simply defines a practical interface of functions for converting the Python code from an input port into a list of ASTs, using the functionality from the `lex+yacc` module.

In a similar way, the `compile` module defines a practical interface of functions for converting lists of ASTs into syntax objects with the compiled code, by calling the `to-racket` method on each AST.

3.1.3 Runtime Modules

The `libpython` module defines a foreign function interface to the functions provided by the Python/C API. Its use will be explained in detail on the next section.

Compiled code contains references to Racket functions and macros, as well as some additional functions which implement Python's primitives. For instance, we define `py-add` as the function which implements the semantics of Python's `+` operator. These primitive functions are defined in the `runtime` module.

3.2 Runtime Implementation using Racket's Foreign Function Interface

For the runtime, we started by following a similar approach to PLT Spy, by mapping Python's data types and primitive functions to the Python/C API. The way we interact with this API, however, is radically different.

On PLT Spy, this was done via the PLT Scheme C API, and therefore the runtime is implemented in C. This entails converting Scheme values into Python objects and vice-versa for each runtime call. Besides the performance issue (described on the Related Work section), this method is cumbersome and lacks portability, since it requires compiling the runtime with a platform-specific C compiler, and to do so each time the runtime is modified.

Instead, we used the Racket Foreign Function Interface (FFI) to directly interact with the foreign data types created by the Python/C API, therefore our runtime is implemented in Racket. These foreign functions are defined on the `libpython` modules, according to their C signatures, and are called by the functions and macros defined on the `runtime` module.

The values passed around correspond to pointers to objects in CPython's virtual machine, but there is sometimes the need to convert them back to Racket data types, so they that can be used in control flow forms like `if`s and `cond`s.

As with PLT Spy, this approach only requires implementing the Python language constructs, because the standard library and other libraries installed on CPython's implementation are readily accessible.

Unfortunately, as we will show in the Performance section, the repetitive use of these foreign functions introduces a huge overhead on our primitive operators, resulting in a very slow implementation.

Additionally, objects allocated with the Python/C API must have their reference counters explicitly decremented, or they will not be garbage collected. This can be solved by attaching a finalizer to each FFI function that allocates a new Python object. This finalizer is responsible for decrementing the object's reference counter when Racket's GC proves that there are no more live references to the Python object. While this solves the garbage collection issue, it entails having another layer of expansive FFI calls, which degrade the runtime performance.

For these reasons, we've tried a second approach, which is described in the following section.

3.3 Runtime Implementation using Racket

Our second approach consists in implementing Python's data model purely in Racket.

In Python, every object has an associated type-object (where every type-object's type is the `type` type-object). A type-object contains a hash table which maps operation names (strings) to the respective functions that type supports (function pointers, in CPython).

As a practical example, in the expression `obj1 + obj2`, the behaviour of the plus operator is determined at runtime, by computing `obj1`'s type-object and looking up the string `__add__` in its hash table. This dynamism in Python allows objects to change behaviour during the execution of a program, simply by adding, modifying of deleting entries from these hash tables, but it also forces an interpreter to constantly lookup these behaviours, contributing to Python's slow performance when compared to other languages.

In CPython, an object's type is stored at the `PyObject` structure. We can use the same strategy in Racket, but there is a nicer alternative. In Racket, one can recognize a value's type through its predicate (`number?`, `string?`, etc.). A Python object's type never changes, so we can directly map basic Racket types into Python's basic types and make their types available through a pattern matching function, which returns the most appropriate type-object, according to the predicates that value satisfies.

This way, we avoid the overhead from constantly wrapping and unwrapping frequently used values from the structures that hold them and it also leads to a cleaner interoperability between Racket and Python code, from the user's perspective. Complex built-in types, such as type-objects, are still implemented through Racket structures.

Comparing it to the FFI approach, this one entails implementing all of Python's standard library in Racket, but, on the other hand, it is much faster and provides reliable memory management of Python's objects, since it does not need to coordinate with another virtual machine.

3.4 Examples

In this section we provide some examples of the current state of the translation between Python and Racket. Note that this is still a work in progress and, therefore, the compilation results of these examples are likely to change in the future.

3.4.1 Fibonacci

Consider the following program in Racket which implements a naive algorithm for computing the Fibonacci function:

```
1  (define (fib n)
2    (cond
3      [(= n 0) 0]
4      [(= n 1) 1]
5      [else (+ (fib (- n 1))
6               (fib (- n 2)))]))
7
8  (fib 30)
```

Its equivalent in Python would be:

```
1  def fib(n):
2    if n == 0: return 0
3    elif n == 1: return 1
4    else: return fib(n-1) + fib(n-2)
5
6  print fib(30)
```

Currently, this code is compiled to:

```
1  (define (:fib :n)
2    (cond
3     ((py-truth (py-eq :n 0)) 0)
4     ((py-truth (py-eq :n 1)) 1)
5     (else (py-add (py-call :fib (py-sub :n 1))
6                   (py-call :fib (py-sub :n 2))))))
7
8  (py-print (py-call :fib 30))
```

Starting with line 1, the first thing one might notice is the colon prefixing the identifiers `fib` and `n`. This has no syntactic meaning in Racket; it is simply a name mangling technique to avoid replacing Racket's bindings with bindings defined in Python. For example, one might set a variable `cond` in Python, which would then be compiled to `:cond` and therefore would not interfere with Racket's built-in `cond`. This also prevents Racket bindings from leaking into Python user code. For instance, the functions `car` and `cdr` will only be available in Python if explicitly imported from Racket.

The functions and macros starting with the `py-` prefix are all defined on the `runtime` module. The functions `py-eq`, `py-add`, and `py-sub` implement the semantics of the Python operators `==`, `+`, and `-`, respectively. The function `py-truth` takes a Python object as argument and returns a Racket Boolean value, `#t` or `#f`, according to Python's semantics for Boolean values. This conversion is necessary because, in Racket, only `#f` is treated as false, while, in Python, the Boolean value false, zero, the empty list and the empty dictionary, among others, are all treated as false when used on the condition of an `if`, `for` or `while` statement. Finally, `py-call` and `py-print` implement the semantics of function calling and the print statement, respectively.

Implementing a runtime function such as `py-add`, with the FFI strategy, is literally as simple as calling two functions from the foreign interface: one for getting the `__add__` attribute from the first operand's type-object and another to call it with the correct arguments. With the Racket runtime strategy, this entails implementing the attribute referencing and method calling semantics. Additionally, to fully cover the semantics of the plus operator, we'll have to implement the `__add__` method for every built-in type that supports it.

The compilation process is independent of the runtime strategy used. Since literals values are pre-computed at compile-time, the only difference in the compilation results are the literals (using the FFI approach, the literals `0`, `1`, `2` and `30` would be foreign pointers to corresponding CPython integer objects). This means that moving from the FFI to the Racket strategy does not entail changing the compiler.

As a final remark for this example, notice that except for the added verboseness, the original Racket code and the compiled code are essentially the same. This is relevant to ensure that DrRacket/Rosetta users that program in Python get a coherent behaviour from DrRacket's step-by-step debugger.

3.4.2 Sieve of Eratosthenes

Consider now a Racket program which implements a variation of the sieve of Eratosthenes, that counts the number of primes below a given number, as shown on listing 1. Its Python equivalent could be implemented as presented on listing 2.

This program presents some other compilation challenges, in order to preserve Python's semantics for binding scopes and control flow.

First of all, in Python we can assign new local variables anywhere, as shown in line 2, while in Racket they have to be declared, e.g., with a `let` form. In Python, only modules, class definitions, function definitions and lambda define a new binding scope. Unlike such languages as C and Java, compound statements (i.e. "blocks") in Python do not define their own scope. Therefore, a variable which is first assigned in a compound statement is also visible outside of it.

This can be implemented by enclosing the body of a function definition inside a `let` form containing all the referenced local variables. This way, Python assignments can be mapped to `set!` forms. Global assignments follow a similar strategy: the variable is first "declared" with a `define` form and then it can be `set!`.

Listing 1 Racket implementation for Sieve of Eratosthenes.

```
1  (define (sieve n)
2    (let ([primes (make-vector n #t)]
3          [counter 0])
4      (for ([i (in-range 2 n)])
5        (when (vector-ref primes i)
6          (set! counter (add1 counter))
7          (for ([j (in-range (* i i) n i)])
8            (vector-set! primes j #f))))
9      counter))
10
11 (sieve 10000000)
```

Listing 2 Python implementation for Sieve of Eratosthenes.

```python
def sieve(n):
  primes = [True] * n
  counter = 0
  for i in range(2,n):
    if primes[i]:
      counter = counter + 1
      for j in range(i*i, n, i):
        primes[j] = False
  return counter

print sieve(10000000)
```

As for Python's `for` statements, Racket provides a `for` macro with similar semantics, however Python's `for` loop allows using `break` and `continue` statements to alter the control flow inside the loop.

We implement it with our own `py-for` macro. It expands to a named `let` which updates the control variables, evaluates the `for`'s body and recursively calls itself, repeating the cycle with the next iteration. A `continue` statement is implemented via calling the named `let` before the end of the body, thus starting a new iteration of the loop, while a `break` statement is handled with escape continuations. Listing 3 shows the final program.

Listing 3 Sieve of Eratosthenes Python version implemented in Racket.

```racket
(define (:sieve :n)
  (let ([:j (void)]
        [:i (void)]
        [:counter (void)]
        [:primes (void)])
    (begin
      (set! :primes (py-mul (make-py-list :True) :n))
      (set! :counter 0)
      (py-for continue43341
       [:i (py-call :range 2 :n)]
       (cond
        ((py-truth (py-index :primes :i))
         (begin
           (set! :counter (py-add :counter 1))
           (py-for continue50281
            [:j (py-call :range (py-mul :i :i) :n :i)]
            (py-set-index :primes :j :False))))
        (else py-None)))
      :counter)))

(py-print (py-call :sieve 10000000))
```

3.5 Performance

In this section we discuss the performance of our implementation and we compare it to the official Python implementation and to the semantically equivalent Racket code.

The charts on Figure 3 compare the running time of these examples for:

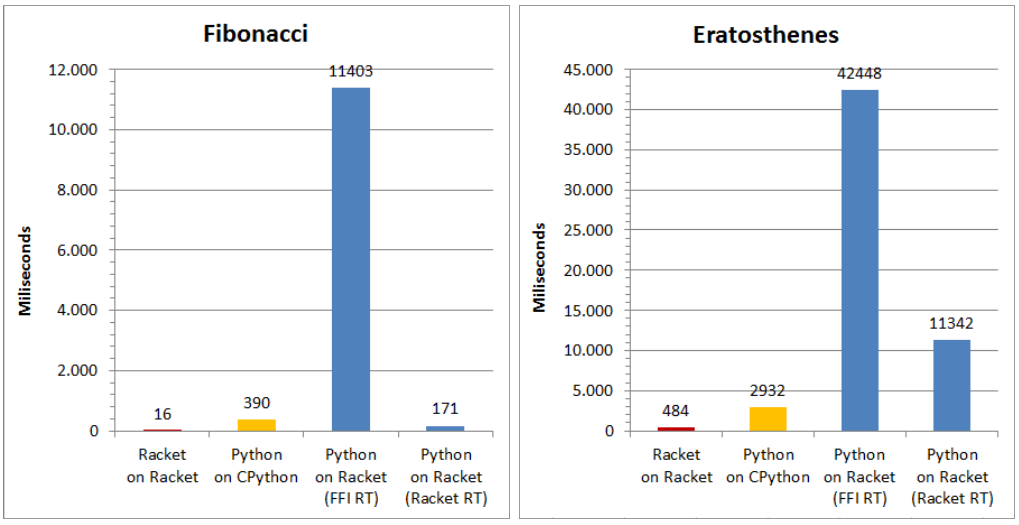


Figure 3 Benchmarks of the Fibonacci and Sieve of Eratosthenes examples.

- Racket code running on Racket;
- Python code running on CPython;
- Compiled Python code running on Racket with the FFI runtime approach;
- Compiled Python code running on Racket with the Racket runtime approach.

These benchmarks were performed on an Intel® Core™ i7 processor at 3.2GHz running under Windows 7. The times below represent the minimum out of 3 samples.

It can be seen that with our first approach (runtime implemented with foreign functions to CPython's virtual machine), Python code running on Racket is currently about 20-30 times slower than the same Python code running on CPython. This is mainly due to the overhead from the FFI calls, which is especially significant since simple operations like an addition entail several FFI calls.

The times presented for this approach do not include the overhead from the finalizers, which severely penalizes execution times. Benchmarks with these examples point to an increase in execution times by a factor between 100% and 150%.

With our second approach (runtime implemented purely on Racket), the Fibonacci example running on Racket is now faster than the same code running on CPython. This can be attributed to Racket's lighter function calls and more efficient primitive operators. We have optimized the use of operators on commonly used types. Since most uses of the `+` and `-` are for numbers, our implementation of these operators tests this case and dispatches the corresponding Racket function for number addition and subtraction, instead of invoking Python's heavier method dispatching mechanism. This is a valid approach, because it is not possible, in Python, to change the predefined semantics of the built-in types.

The Sieve of Eratosthenes example still runs slower than in CPython, but it's performance is quite acceptable for our current goals.

4 Conclusions

There is a need for an implementation of Python for the Rosetta community, and, more specifically, for Racket users. This implementation must be able to interact with Racket libraries and should be as close as possible to other state-of-the-art implementations in terms of performance.

Our solution follows a traditional compiler's approach, as a pipeline of scanner, parser and code generation. Python source-code is, thus, compiled to equivalent Racket source-code. This Racket source-code is then handled by Racket's bytecode compiler, JIT compiler and interpreter.

We have presented two approaches for the runtime implementation. The first one makes use of Racket's Foreign Interface and the Python/C API handle Python objects in CPython's virtual machine. This allows our implementation to effortlessly support all of Python's standard library and even third-party libraries written in C. On the other hand, it suffers from bad performance (at least one order of magnitude slower than CPython).

It is worth noting, however, that this approach can be used for quickly implementing a runtime for other interpreted languages, provided that they have an API which allows foreign function access to their functionality. Such languages include Ruby (using the Ruby C API [19]), Lua (using the Lua C API [7]) and SQL (using the MySQL C API [26] or PostgreSQL's `libpq` [14]).

Implementing a language this way would just require building a parser and a compiler which handles the program's control flow, since the language's data model and libraries would be handled by the API.

Our second approach consists in implementing Python's data model and functions in Racket. This leads to a greater effort in order to implement all of Python's standard library, but allows for a better integration with Racket code, and a better performance, currently standing at about 4 times slower than CPython.

We will be following our second approach, but we may offer support for accessing third-party libraries and unimplemented standard library modules using the features from our first approach.

Some of Python's common expressions and control flow statements have been already implemented, allowing for the successful compilation of two examples: the Fibonacci sequence and an implementation of the sieve of Eratosthenes.

In the future, we plan on implementing the remaining Python features and work on the integration between Python and Racket code.

Acknowledgements. This work was partially supported by Portuguese national funds through FCT under contract Pest-OE/EEI/LA0021/2013 and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

References

1 David Beazley. Understanding the Python GIL. In *PyCON Python Conference*, Atlanta, Georgia, 2010.

2 David M Beazley. *Python Essential Reference*. Addison-Wesley Professional, 2009.

3 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

4 Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer Berlin Heidelberg, 1997.

5 Matthew Flatt and Robert Bruce Findler. *The Racket Guide*, 2013.

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

7 Roberto Ierusalimschy. *Programming in lua*, chapter An Overview of the C API. Roberto Ierusalimschy, 2006.

8 Ironclad – Resolver Systems. `http://www.resolversystems.com/products/ironclad/`. [Online; retrieved on January 2014].

9 Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Munoz Soto, and Victor Ng. *The definitive guide to Jython*. Springer, 2010.

10 Differences between CPython and Jython. `http://jython.sourceforge.net/archive/21/docs/differences.html`. [Online; retrieved on December 2013].

11 José Lopes and António Leitão. Portable generative design for CAD applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.

12 Philippe Meunier and Daniel Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

13 Microsoft Corporation. *IronPython .NET Integration documentation*. `http://ironpython.net/documentation/`. [Online; retrieved on January 2014].

14 Bruce Momjian. *PostgreSQL: introduction and concepts*, chapter C Language Interface (LIBPQ).

15 Benjamin Peterson. Pypy. *The Architecture of Open Source Applications*, 2:279–290.

16 PyPy compatibility. `http://pypy.org/compat.html`. [Online; retrieved on December 2013].

17 PyPy speed center. `http://speed.pypy.org/`. [Online; retrieved on December 2013].

18 Stefan Richthofer. JyNI – using native CPython-extensions in Jython. In *EuroSciPi 2013*, Brussels, Belgium, 2013.

19 Muriel Salvan. The Ruby C API – basics. `http://blog.x-aeon.com/2012/12/13/the-ruby-c-api-basics/`. [Online; retrieved on March 2014].

20 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *ACM SIGPLAN Notices*, 46(6):132–141, 2011.

21 Peter Tröger. Python 2.5 virtual machine. `http://www.troeger.eu/files/teaching/pythonvm08.pdf`, April 2008. [Lecture at Blekinge Institute of Technology].

22 Guido van Rossum and Fred L. Drake. *Extending and embedding the Python interpreter*. Centrum voor Wiskunde en Informatica, 1995.

23 Guido van Rossum and Fred L. Drake. *An introduction to Python*. Network Theory Ltd., 2003.

24 Guido van Rossum and Fred L. Drake. *The Python Language Reference*. Python Software Foundation, 2010.

25 Guido van Rossum and Fred L. Drake Jr. *Python/C API reference manual*, chapter Thread State and the Global Interpreter Lock. Python Software Foundation, 2002.

26 Michael Widenius and David Axmark. *MySQL reference manual: documentation from the source*, chapter MySQL C API. O'Reilly Media, Inc., 2002.