# An Implementation of Python for 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 descendent of Scheme that is widely used as a first language for teaching computer science. To this end, Racket provides DrRacket, a simple but pedagogic IDE. On the other hand, Python is becoming increasingly popular in a variety of areas, most notably among novice programmers. This paper presents an implementation of Python for Racket which allows programmers to use DrRacket with Python code, as well as adding Python support for other DrRacket based tools. Our implementation also allows Racket programs to take advantage of Python libraries, thus significantly enlarging the number of usable libraries in Racket.

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

The first strategy provides immediate support for Python's standard library and existing third-party libraries. The second strategy requires a Racket-based reimplementation of all of Python's features, but provides native interoperability between Python and Racket code.

Our experimental results show that the second strategy far outmatches the first in terms of speed. Furthermore, it is more portable since it has no dependencies on Python's virtual machine.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

## General Terms

Languages

## Keywords

Python; Racket; Language implementations; Compilers

## 1. INTRODUCTION

The Racket programming language is a descendent of Scheme, a language that is well-known for its use in introductory programming courses. Racket comes with DrRacket, a pedagogic IDE [2], used in many schools around the world, as it provides a simple and straightforward interface aimed at inexperienced programmers. Racket provides different language levels, each one supporting more advanced features, that are used in different phases of the courses, allowing students to benefit from a smoother learning curve. Furthermore, Racket and DrRacket support the development of additional programming languages [13].

More recently, the Python programming language is being promoted as a good replacement for Scheme (and Racket) in computer science courses. 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. 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 alternative implementations such as Jython (written in Java) and IronPython (written in C#).

According to Peter Norvig [11], Python is an excellent language for pedagogical purposes and is easier to read than Lisp for someone with no experience in either language. He describes Python as a dialect of Lisp with infix syntax, as it supports all of Lisp's essential features except macros. Python's greatest downside is its performance. Compared to, e.g., Common Lisp, Python is around 3 to 85 times slower for most tasks.

Despite its slow performance, Python is becoming an increasingly popular programming language on many areas, due to its large standard library, expressive syntax and focus on code readability.

In order to allow programmers to easily move between Racket and Python, we are developing an implementation of Python for Racket, that preserves the pedagogic advantages of DrRacket's IDE and provides access to the countless Python libraries.

As a practical application of this implementation, we are developing Rosetta [8], a DrRacket-based IDE, aimed at architects and designers, that promotes a programming-based approach for modelling three-dimensional structures. Although Rosetta's modelling primitives are defined in Racket, Rosetta integrates multiple programming languages, including Racket, JavaScript, and AutoLISP, with multiple computer-aided design applications, including AutoCAD and Rhinoceros 3D.

Our implementation adds Python support for Rosetta, allowing Rosetta users to program in Python. Therefore, this implementation must support calling Racket code from Python, using Racket as an interoperability platform. Being able to call Python code from Racket is also an interesting feature for Racket developers, by allowing them to benefit from the vast pool of existing Python libraries.

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, written in the C programming language, 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 extensions 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 [15].

CPython's virtual machine is a simple stack-based machine, where the byte codes operate on a stack of `PyObject` pointers [14]. At runtime, every Python object has a corresponding `PyObject` instance. A `PyObject` contains a reference counter, used for garbage collection, and a pointer to a `PyTypeObject`, which specifies the object's type (and is also a `PyObject`). 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 makes use of memory pools for small memory requests. Additionally, it also pre-allocates commonly used immutable objects (such as the integers from -5 to 256), so that new references will point to these instances instead of allocating new ones.

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 to an object is discarded, its reference counter is decremented. When it reaches zero, the object's finalizer is called and the space is reclaimed.

Reference counting, however, does not work well with reference cycles [17, ch. 3.1]. Consider the example of a list containing 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.

### 2.2 Jython and IronPython

Jython is an alternative 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).

Jython programs cannot use extension modules written for CPython, but they can import Java classes, using the same syntax that is used for importing Python modules. It is worth mentioning that since Clojure targets the JVM, Jython makes it possible to import and use Clojure libraries from Python and vice-versa [5]. There is also work being done by a third-party [12] to integrate CPython module extensions with Jython, through the use of the Python/C API. This would allow popular C-based libraries such as NumPy and SciPy to be used with Jython.

Garbage collection in Jython is performed by the JVM and does not suffer from the issues with reference cycles that plague CPython [7, p. 57]. In terms of speed, 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.

IronPython is another alternative implementation of Python, this one for Microsoft's Common Language Infrastructure (CLI). It is written in C# and was first released in 2006. Similarly to what Jython does for the JVM, IronPython compiles Python source-code to CLI bytecode, which can be run on the .NET framework. It claims to be 1.8 times faster than CPython on `pystone`, a Python benchmark for showcasing Python's features.

IronPython provides support for importing .NET libraries and using them with Python code [10]. There is also work being done by a third-party in order to integrate CPython module extensions with IronPython [6].

### 2.3 CLPython

CLPython (not to be confused with CPython, described above) is yet another Python implementation, written in Common Lisp. Its development was first started in 2006, but stopped in 2013. It supports six Common Lisp implementations: Allegro CL, Clozure CL, CMU Common Lisp, ECL, LispWorks and SBCL [1]. Its main goal was to bridge Python and Common Lisp development, by allowing access to Python libraries from Lisp, access to Lisp libraries from Python and mixing Python and Lisp code.

CLPython compiles Python source-code to Common Lisp code, i.e. a sequence of s-expressions. These s-expressions can be interpreted or compiled to `.fasl` files, depending on the Common Lisp implementation used. Python objects are represented by equivalent Common Lisp values, whenever possible, and CLOS instances otherwise.

Unlike other Python implementations, there is no official performance comparison with a state-of-the-art implementation. Our tests (using SBCL with Lisp code compilation) show that CLPython is around 2 times slower than CPython on the `pystone` benchmark. However it outperforms CPython on handling recursive function calls, as shown by a benchmark with the Ackermann function.

## 2.4 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 [9].

PLT Spy's runtime library is written in C and extended to Scheme via the PLT Scheme C API. It implements Python's built-in types and operations by mapping them to CPython's 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.

## 2.5 Comparison

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

| | Platform(s) targeted | Speedup (vs. CPython) | Std. library support |
|---|---|---|---|
| **CPython** | CPython's VM | 1× | Full |
| **Jython** | JVM | ∼ 1× | Most |
| **IronPython** | CLI | ∼ 1.8× | Most |
| **CLPython** | Common Lisp | ∼ 0.5× | Most |
| **PLT Spy** | Scheme | ∼ 0.001× | Full |

**Table 1: Comparison between implementations**

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. Unfortunately, there is a considerable performance cost that results from the repeated conversion of data from Scheme's internal representation to CPython's internal representation.

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

Taking this into consideration, we developed a Python implementation for Racket that we present in the next section.

## 3. SOLUTION

Our proposed solution consists of two compilation phases: (1) Python source-code is compiled to Racket source-code and (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 semantically 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, inlining, 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 Racket, therefore our implementation effort relies only on a source-to-source compiler from Python to Racket.

### 3.1 General Architecture

**Fig. 1** summarises the dependencies between the different Racket modules of the proposed solution. The next paragraphs provide a more detailed explanation of these modules.

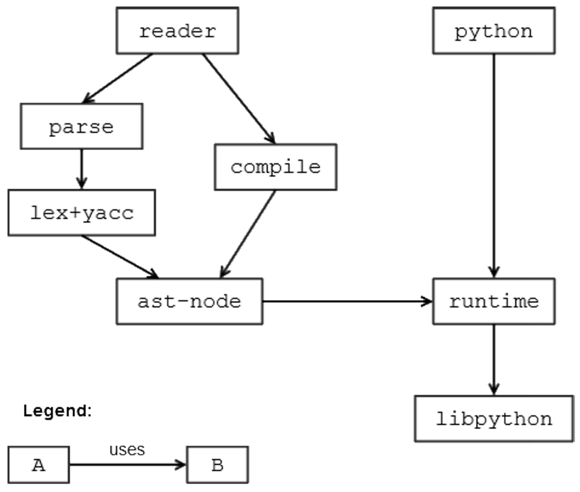

**Figure 1: Dependencies between modules. The arrows indicate that a module uses functionality that is defined on the module it points to.**

### 3.1.1 Racket Interfacing

A Racket file usually starts with the line `#lang <language>` to specify which language is being used (in our case, it will be `#lang python`). The entry-point for a `#lang` is at the `reader` module, visible at the top of **Fig. 1**. This module must provide the functions `read` and `read-syntax` [4, ch. 17.2].

The `read-syntax` function takes the name of the source file and an input port as arguments and returns a list of syntax objects, which correspond to the Racket code compiled from the input port. It uses the `parse` and `compile` modules to do so.

Syntax objects [4, ch. 16.2.1] are Racket's built-in data type for representing code. They contain the quoted form of the code (an s-expression), source location information (line number, column number and span) and lexical-binding information. By keeping the original source location information on every syntax object generated by the compiler, DrRacket can map each compiled s-expression to its corresponding Python code. This way, DrRacket's features for Racket code will also work for Python. Such features include the syntax checker, debugger, displaying source location for errors, tacking and untacking arrows for bindings and renaming variables.

### 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 `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 generating a syntax object with the compiled code and respective source location. 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 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.

Finally, the `python` module simply provides everything defined at the `runtime` module, along with all the bindings from the `racket` language. Thus, every identifier needed for the compiled code is provided by the `python` module.

## 3.2 Runtime Implementation using FFI

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 lacks portability and is somewhat cumbersome for development, since it requires compiling the runtime module with a platform specific C compiler, and to do so each time this module 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 can be used as conditions in flow control 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 significant overhead on our primitive operators, resulting in a very slow implementation.

Another issue is that the Python objects allocated on CPython's VM must have their reference counters explicitly decremented or they will not be garbage collected. This issue can be solved by attaching a Racket finalizer to every FFI function that returns a new reference to a Python object. This finalizer will decrement the object's reference counter whenever Racket's GC proves that there are no more live references to the Python object. On the other hand, this introduces another significant performance overhead.

## 3.3 Runtime Implementation using Racket

Our second approach is a pure Racket implementation of Python's data model. 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 a much faster implementation and provides reliable memory management of Python's objects, since it does not need to coordinate with another virtual machine.

### 3.3.1 Object Model

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 list of base types and a hash table which maps operation names (strings) to the functions that type supports (function pointers, in CPython).

As a practical example, in the expression `obj1 + obj2`, the behaviour of the plus operator depends on the type of its operands. If `obj1` is a number this will be the addition operator. If it is a string, this will be a string concatenation. Additionally, a user-defined class can specify another behaviour for the plus operator by defining the method `__add__`. This is typically done inside a class definition, but can also be done after the class is defined, through reflection.

CPython stores each object's type as a pointer in the `PyObject` structure. Since an object's type is not known at compile-time, method dispatching must be done at runtime, by obtaining `obj1`'s type-object and looking up the function that is mapped by the string `__add__` on its hash table. If there is no such entry, the search continues on that type-object's base types.

While the same mechanics would work in Racket, there is room for optimization. In Racket, one can recognize a value's type through its predicate (`number?`, `string?`, etc.). In Python, a built-in object's type is not allowed to change, so we can directly map basic Racket types into Python's basic types. Their types are computed through a pattern matching function, which returns the most appropriate type-object, according to the predicates that value satisfies. Complex built-in types are still implemented through Racket structures (which include a reference to the corresponding type-object).

This way, we avoid the overhead from constantly wrapping and unwrapping frequently used values from the structures that hold them. Interoperability with Racket data types is also greatly simplified, eliminating the need to wrap/unwrap values when using them as arguments or return values from functions imported from Racket.

There is also an optimization in place concerning method dispatching. Despite the ability to add new behaviour for operators in user-defined classes, a typical Python program will mostly use these operators for numbers (and strings, in some cases). Therefore, each operator implements an early dispatch mechanism for the most typical argument types, which skips the heavier dispatching mechanism described above. For instance, the plus operator is implemented as such:

```
(define (py-add x y)
  (cond
    [(and (number? x) (number? y)) (+ x y)]
    [(and (string? x) (string? y)) (string-append x y)]
    [else (py-method-call x "__add__" y)]))
```

### 3.3.2 Importing Modules

In Python, files can be imported as modules, which contain bindings for defined functions, defined classes and global assignments. Unlike in Racket, Python modules are first-class citizens. There are 3 ways to import modules in Python: (1) the `import <module>` syntax, which imports <module> as a module object whose bindings are accessible as attributes; (2) the `from <module> import <binding>` syntax, which only imports the declared <binding> from <module>; (3) the `from <module> import *` syntax, which imports all bindings from <module>.

To implement the first syntax, we make use of `module->exports` to get a list of the bindings provided by a module and `dynamic-require` to import each one of them and store them in a new module object. The other two syntaxes are semantically similar to Racket's importing model and, therefore, are implemented with `require` forms.

This implementation of the import system was designed to allow importing both Python and Racket modules. We have come up with a slightly different syntax for referring to Racket modules. They are specified as a string literal containing a Racket module path (following the syntax used for a `require` form [3, ch. 3.2]).

This way we support importing bindings from the Racket library, Racket files or packages hosted on PLaneT (Racket's centralized package distribution system), using any of the Python importing syntaxes mentioned above. The following example shows a way to access the Racket functions `cons`, `car` and `cdr` in a Python program.

```
#lang python
import "racket" as racket

def add_cons(c):
  return racket.car(c) + racket.cdr(c)

c1 = racket.cons(2, 3)
c2 = racket.cons("abc", "def")
```

```
> add_cons(c1)
5
> add_cons(c2)
"abcdef"
```

Since the second and third syntaxes above map to `require` forms (which are evaluated before macro expansion), it is also possible to use Racket-defined macros with Python code.

Predictably, importing Python modules into Racket programs is also possible and straightforward. Function definitions, class definitions and top-level assignments are `define`'d and `provide`'d in the compiled Racket code, therefore they can be `require`'d in Racket.

### 3.3.3 Class Definitions

A class definition in Python is just syntactic sugar for defining a new type-object. Its hash table will contain the variables and methods defined within the class definition. Therefore, an instance of a class is an object like any other, whose type-object is its class. The main distinction is that an instance of a class also contains its own hash table, where its attributes are mapped to their values.

### 3.3.4 Exception Handling

Both Python and Racket support exceptions in a similar way. In Python, one can only raise objects whose type derives from `BaseException`, while in Racket, any value can be raised and caught.

In Python, exceptions are raised with the `raise` statement and caught with the `try...except` statement (with optional else and finally clauses). Their semantics can be implemented with Racket's raise and with-handlers forms, respectively. The latter expects an arbitrary number of pairs of predicate and procedure. Each predicate is responsible for recognizing a specific exception type and the procedure is responsible for handling it.

The exceptions themselves can be implemented as Racket exceptions. In fact, some of Python's built-in exceptions can be defined as their equivalents in Racket, for added interoperability. For instance, Python's ZeroDivisionError can be mapped to Racket's exn:fail:contract:divide-by-zero and Python's NameError is mapped to Racket's exn:fail:contract:variable.

## 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 may change in the future.

### 4.1 Ackermann

Consider the following program in Racket which implements the Ackermann function and calls it with arguments $m = 3$ and $n = 9$:

```
(define (ackermann m n)
  (cond
   [(= m 0) (+ n 1)]
   [(and (> m 0) (= n 0)) (ackermann (- m 1) 1)]
   [else (ackermann (- m 1) (ackermann m (- n 1)))]))

(ackermann 3 9)
```

Its equivalent in Python would be:

```
def ackermann(m,n):
    if m == 0: return n+1
    elif m > 0 and n == 0: return ackermann(m-1,1)
    else: return ackermann(m-1, ackermann(m,n-1))

print ackermann(3,9)
```

Currently, this code is compiled to:

```
(provide :ackermann)
(define-py-function :ackermann with-params (m n)
  (lambda (:m :n)
    (cond
      [(py-truth (py-eq :m 0))
       (py-add :n 1)]
      [(py-truth (py-and (py-gt :m 0) (py-eq :n 0)))
       (py-call :ackermann (py-sub :m 1) 1)]
      [else
       (py-call
        :ackermann
        (py-sub :m 1)
        (py-call :ackermann :m (py-sub :n 1)))])))

(py-print (py-call :ackermann 3 9))
```

The first thing one might notice is the colon prefixing the identifiers ackermann, m 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.

The (define-py-function ... with-params ...) macro builds a function structure, which is essentially a wrapper for a lambda and a list of the argument names. The need to store a function's argument names arises from the fact that in Python a function can be called both with positional or keyword arguments. A function call without keyword arguments is handled by the py-call macro, which simply expands to a traditional Racket function call. If the function is called with keyword arguments, this is handled by py-call/keywords, which rearranges the arguments' order at runtime.

This way, we can use the same syntax for calling both Python user-defined functions and Racket functions. On the other hand, since the argument names are only stored with Python user-defined functions, it is not possible to use keyword arguments for calling Racket functions.

The functions/macros py-eq, py-and, py-gt, py-add and py-sub are defined on the runtime module and 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, the function py-print implements the semantics of the print statement.

### 4.2 Mandelbrot

Consider now a Racket program which defines and calls a function that computes the number of iterations needed to determine if a complex number $c$ belongs to the Mandelbrot set, given a limited number of *limit* iterations.

```
(define (mandelbrot limit c)
  (let loop ([i 0]
             [z 0+0i])
    (cond
     [(> i limit) i]
     [(> (magnitude z) 2) i]
     [else (loop (add1 i)
                 (+ (* z z) c))])))

(mandelbrot 1000000 .2+.3i)
```

Its Python equivalent could be implemented like such:

```
def mandelbrot(limit, c):
    z = 0+0j
    for i in range(limit+1):
        if abs(z) > 2:
            return i
        z = z*z + c
    return i+1

print mandelbrot(1000000, .2+.3j)
```

This program demonstrates some features which are not straightforward to map in Racket. For example, in Python we can assign new local variables anywhere, as shown in line 2, while in Racket they become parameters of a named `let` form.

Another feature, present in most programming languages but not in Racket, is the `return` keyword, which immediately returns to the point where the function was called, with a given value. On the former example, all returns were tail statements, while on this one we have an early return, on line 5.

The program is compiled to:

```
 1  (provide :mandelbrot)
 2  (define-py-function :mandelbrot with-params (limit c)
 3    (lambda (:limit :c)
 4      (let ([:i (void)]
 5            [:z (void)])
 6        (let/ec return9008
 7          (set! :z (py-add 0 0))
 8          (py-for continue9007
 9            [:i (py-call :range (py-add :limit 1))]
10            (begin
11              (cond
12                [(py-truth (py-gt (py-call :abs :z) 2))
13                 (return9008 :i)]
14                [else py-None])
15              (set! :z (py-add (py-mul :z :z) :c))))
16          (return9008 (py-add :i 1))))))
17
18  (py-print
19   (py-call :mandelbrot 1000000 (py-add 0.2 0+0.3i)))
```

You will notice the `let` form on lines 4-5. The variables `:i` and `:z` are declared with a void value at the start of the function definition, allowing us to simply map Python assignments to `set!` forms.

Early returns are implemented as escape continuations, as seen on line 6: there is a `let/ec` form (syntactic sugar for a `let` and a `call-with-escape-continuation`) wrapping the body of the function definition. With this approach, a return statement is as straightforward as calling the escape continuation, as seen on line 13.

Finally, `py-for` is a macro which implements Python's for loop. 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. Note that calling this named `let` has the same semantics as a `continue` statement.

In fact, although there was already a `for` form in Racket with similar semantics as Python's, the latter allows the use of `break` and `continue` as flow control statements. The `break` statement can be implemented as an escape continuation and `continue` is implemented by calling the named `let`, thus starting a new iteration of the loop.

## 5. PERFORMANCE

The charts on **Fig. 2** compare the running time of these examples for:

- **(a)** Racket code running on Racket;
- **(b)** Python code running on CPython;
- **(c.1)** Python code running on Racket with the FFI runtime approach, without finalizers
- **(c.2)** Python code running on Racket with the FFI runtime approach, with finalizers for garbage collecting Python objects
- **(d.1)** Python code running on Racket with the pure Racket runtime approach
- **(d.2)** Python code running on Racket with the pure Racket runtime approach, using early dispatch for operators

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

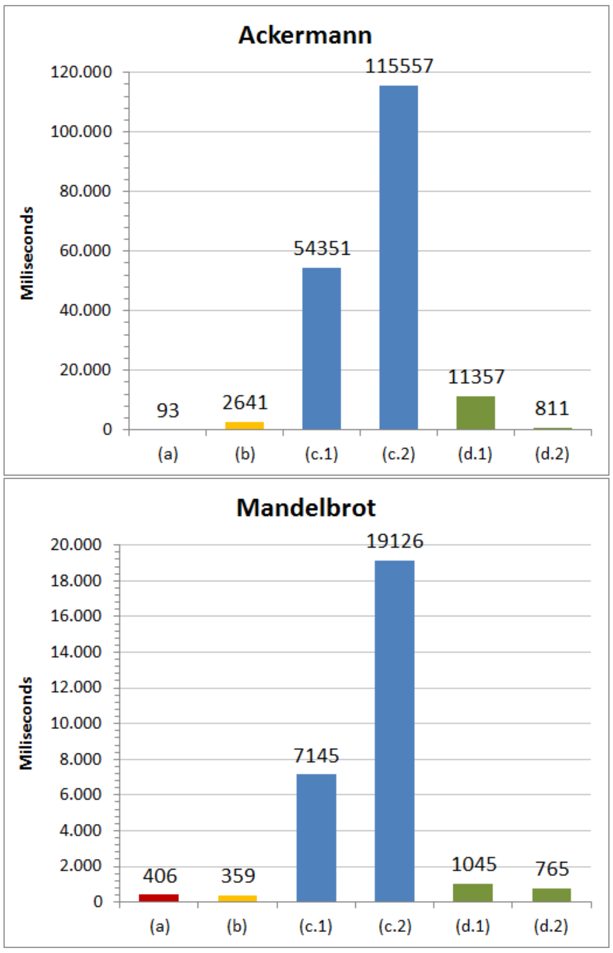

**Figure 2: Benchmarks of the Ackermann and Mandelbrot examples**

The Racket implementation of the Ackermann example is about 28 times faster than Python's implementation, but the Mandelbrot example's implementation happens to be slightly slower than Python's. This is most likely due to

Racket's lighter function calls and operators, since the Ackermann example heavily depends on them.

Since the FFI based runtime uses CPython's primitives, we have to endure with sluggish foreign function calls for every Python operation and we also cannot take advantage of Racket's lightweight mechanics, therefore the same Python code runs about 20 times slower on our implementation than in CPython, for both examples. This figure more than doubles if we consider the use of finalizers, in order to avoid a memory leak.

Moving to a pure Racket runtime yielded a great improvement over the FFI runtime, since it eliminated the need for foreign function calls, synchronizing garbage collection with another virtual machine and type conversions. With this transition, both examples run at around 3 to 4 times slower than in CPython, which is very tolerable for our goals.

Optimizing the dispatching mechanism of operators for common types further led to huge gains in the Ackermann example pushing it below the running time for CPython. The Mandelbrot example is still slower than in CPython, but nonetheless it has also benefited from this optimization.

## 6. CONCLUSIONS

A Racket implementation of Python would benefit Racket developers giving them access to Python's huge standard library and the ever-growing universe of third-party libraries, as well as Python developers by providing them with a pedagogic IDE in DrRacket. To be usable, this implementation must allow interoperability between Racket and Python programs and should be as close as possible to other state-of-the-art implementations in terms of performance.

Our solution tries to achieve these qualities by compiling Python source-code to semantically equivalent Racket source-code, using a traditional compiler's approach: a pipeline of scanner, parser and code generation. This Racket source-code is then handled by Racket's bytecode compiler, JIT compiler and interpreter.

We have come up with two alternative solutions for implementing Python's runtime semantics in Racket. The first one consists of using Racket's Foreign Interface and the Python/C API to manipulate Python objects in Python'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).

Our second approach consists of implementing Python's data model and standard library purely in Racket. This leads to a greater implementation effort, but offers a greater performance, currently standing at around the same speed as CPython, depending on the application. Additionally, it allows for a better integration with Racket code, since many Python data types are directly mapped to Racket data types.

Our current strategy consists of implementing the language's essential features and core libraries using the second approach (for performance and interoperability). Future efforts may include developing a mechanism to import modules from CPython through the FFI approach, in a way that is compatible with our current data model.

## 7. 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.

## 8. REFERENCES

[1] W. Broekema. CLPython - an implementation of Python in Common Lisp. `http://common-lisp.net/project/clpython/`. [Online; retrieved on March 2014].

[2] 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.

[3] M. Flatt. *The Racket Reference*, 2013.

[4] M. Flatt and R. B. Findler. *The Racket Guide*, 2013.

[5] E. Franchi. Interoperability: from Python to Clojure and the other way round. In *EuroPython 2011*, Florence, Italy, 2011.

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

[7] J. Juneau, J. Baker, F. Wierzbicki, L. M. Soto, and V. Ng. *The definitive guide to Jython*. Springer, 2010.

[8] J. Lopes and A. 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.

[9] P. Meunier and D. Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

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

[11] P. Norvig. Python for Lisp programmers. `http://norvig.com/python-lisp.html`. [Online; retrieved on March 2014].

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

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

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

[15] G. van Rossum and F. L. Drake. *Extending and embedding the Python interpreter*. 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*. Python Software Foundation, 2010.