An Implementation of Python for DrRacket

Pedro Palma Ramos

Instituto Superior Técnico, Universidade de Lisboa
pedropramos@tecnico.ulisboa.pt
http://tecnico.ulisboa.pt/

Abstract. The Python programming language is becoming increasingly popular in a variety of areas, most notably among novice programmers. This paper presents an implementation of Python for DrRacket which allows Python programmers to use DrRacket's features with Python code, as well as adding Python support for DrRacket based tools, such as Rosetta. The suggested approach involves compiling Python code into equivalent Racket code. The compiled code makes use of a runtime library that runs on top of Python's virtual machine, therefore using its data types and primitive operations. This approach allows full support for Python's libraries, but intermediate results have evidenced issues with garbage collection and a slowdown by at least one order of magnitude when compared to Python's reference implementation. Future efforts include reducing the dependency on Python's virtual machine by implementing most of its primitives over Racket data types.

Keywords: Python; Racket; DrRacket; Programming languages; Compilers; Rosetta

1 Introduction

There is an increasing need for architects and designers 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. It is, thus, increasingly important for architects and designers to master programming techniques.

Although most computer-aided design (CAD) applications provide programming languages for this end, programs written in these languages have very limited portability: given that each CAD application provides its own specific language and functionality, a program written for one CAD application cannot be used on other CAD applications. On the other hand, these are not pedagogical programming languages and most of them are poorly designed or obsolete.

DrRacket (formerly known as DrScheme) is an integrated development environment (IDE) originally meant for the Racket programming language, a dialect of LISP and a descendant of Scheme.[1] Unlike such IDEs as Eclipse or Microsoft Visual Studio, DrRacket provides a simple and straightforward interface aimed at inexperienced programmers (**Fig. 1**).

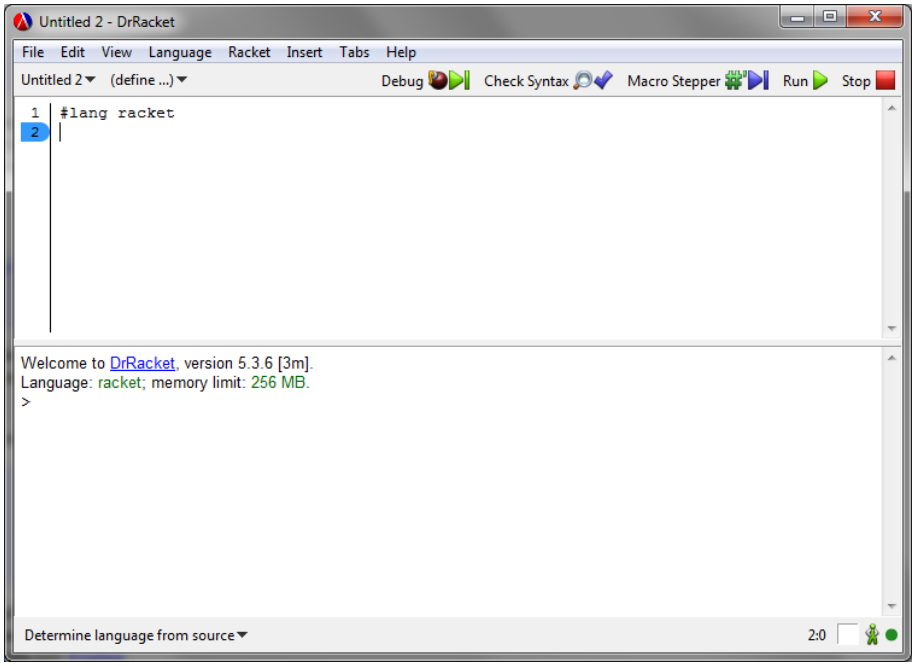

Fig. 1. DrRacket's graphical user interface

DrRacket was designed as a pedagogic environment[2] and has been used in introductory programming courses in many schools around the world. Additionally, DrRacket is not limited to Racket development, as it also supports the development and extension of other programming languages.[3] A number of programming languages have been implemented in Racket such as Algol, Datalog, JavaScript and other dialects of LISP.

Rosetta is an extensible IDE for generative design, based on DrRacket and targeted at architects and designers.[5] It seeks to answer the portability problem by allowing the development of programs in different programming languages which are then portable across different CAD applications. This is achieved by (1) an abstraction layer that allows portable and transparent access to several different CAD applications; (2) back-ends that translate the abstraction layer into different CAD applications; (3) front-end programming languages in which users write the generative design programs; and (4) an intermediate programming language, Racket, that encompasses the language constructs essential for geometric modelling and that is used as a compilation target for the front-ends.

Rosetta users can program directly in Racket or in any other programming language implemented as a front-end. Currently Rosetta supports front-ends for AutoLISP, JavaScript and RosettaFlow (a graphical language inspired in Grasshopper). AutoLISP and JavaScript were chosen precisely because they have

been used for generative design. More recently, the language Python has emerged as a good candidate for this area of application.

In this paper we propose an implementation of the Python programming language for DrRacket.

Python is a very high-level, interpreted, dynamically typed programming language.[9, 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 implementations such as Jython (written in Java), PyPy (written in Python) and IronPython (targeting the .NET Framework).

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

In the next section, we address the objectives for this thesis work. Section 3 explores some related Python implementations. Section 4 describes the architecture of the proposed solution, justifies our major decisions and showcases the performance of our current implementation with benchmarks. Section 5 explains how our results will be evaluated and we conclude with Section 6.

2 Objectives

The main objective of this thesis work is to develop a correct and efficient implementation of the Python language for DrRacket. It will allow for Python developers to use features of the DrRacket IDE such as syntax highlighting, debugger and profiler, as well as developing programs or libraries that mix Python and Racket code. Additionally, it will allow architects and designers to use it as a front-end programming language for Rosetta.

Our main focus will be on exploring efficient methods for mapping Python's semantics on Racket's features. This includes:

- Python's object-based data model
- Built-in data types such as the numeric tower, lists and dictionaries
- Built-in operators such plus (+), greater than (>), etc.
- Function and class definitions
- Bindings and scopes
- Flow control statements such as `return`, `break`, `continue` and `yield`
- Exception handling
- Importing modules

3 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.

3.1 CPython

CPython, written in the C programming language, has been the reference implementation of Python since its first release and is maintained by the Python Software Foundation. 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.[6]

3.1.1 Object Representation

The virtual machine is a simple stack machine[7], where the byte codes operate on a stack of `PyObject` pointers.

At runtime, every Python object has a corresponding `PyObject` instance. A `PyObject` contains a pointer to a reference count, used for garbage collecting, and a `PyTypeObject`, which indicaes the objects 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 counter this, there is a special performance optimization: only requests larger than 256 bytes are handled by `malloc`, the C standard allocator, while smaller ones are handled with memory pools.

In addition to this, 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.

3.1.2 Garbage collection

The traditional garbage collection scheme of following references to track reachable objects does not work in CPython, because extension modules make it impossible to determine the root objects.

Instead, garbage collection is performed through reference counting. Whenever a new Python object is allocated or whenever a new reference to it is made, its reference count is incremented. When its reference is no longer needed, the 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.[8, ch. 3.1] Consider the example of a list appending itself. When its reference goes out of scope, its counter is decremented, however the cyclic reference inside the list is still present, so the reference count will never reach zero and the list will not be garbage collected, even though it's already unreachable.

This is still an open problem in CPython and for this reason, cyclic references are not encouraged.

3.1.3 Threading

Threading in CPython is implemented over real OS threads, however CPython enforces a global interpreter lock (GIL), which prevents more than one thread running interpreted code at the same time. This is necessary because the reference counting system (described above) is not thread-safe.[10] If two threads attempt to increment an object's reference count simultaneously, it would be possible for this count to be erroneously incremented only once.

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.[11] Therefore, the use of threads is only recommended for I/O tasks.[12, p. 444]

Note that the GIL is a feature of CPython and not of the Python language. It is not present in other implementations such as Jython or IronPython.

3.2 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).

3.2.1 Implementation Differences

There are some aspects of Python's semantics in which Jython's implementation differs from CPython's.[13] 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 differences from the one implemented in CPython, as some of the C-based modules have been rewritten in Java.

3.2.2 Java Integration

Jython programs cannot use extension modules written for CPython, but they can import Java classes, the same way any other Python module can be imported.

There is work being done by a third-party[14] to integrate CPython module extensions with Jython, through the use of the Python/C API. This would allow C-based libraries such as NumPy and SciPy to be used with Jython.

3.2.3 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.[15, p. 57] Furthermore, there is no global interpreter lock, so threads can take advantage of multi-processor architectures for CPU-intensive tasks.[15, 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.

3.3 IronPython

IronPython 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 follows an architecture similar to CPython's, with a scanner, parser, bytecode generator and a support library, all written in C#.[16]

3.3.1 .NET Integration

IronPython provides support for importing .NET libraries and using them with Python code.[17] As it happened with Jython, there is work being done by a third-party in order to integrate CPython module extensions with IronPython.[18]

3.3.2 Performance

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.[19]

3.4 PyPy

PyPy is yet another Python implementation, written in RPython, a restricted subset of Python. 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.[20]

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.[21]

PyPy actually includes two very distinct modules:[22]

- The Python interpreter, written in RPython;
- The RPython translation toolchain.

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.

3.4.1 Interpreter

Like the implementations mentioned before, the interpreter converts the 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 interpreter uses an abstraction called object spaces, commonly abbreviated to *objspaces*. An objspace encapsulates the knowledge needed to represent and manipulate a specific Python data type. This allows the interpreter to treat Python objects as black boxes, generating the same code for each operation, without the need to inspect the types of the operands. The actual behaviour for each operation is delegated to a method of the objspace.

Besides enforcing a clean separation between structure and behaviour, this strategy also supports having multiple implementations of a specific data type, which allows for the most efficient one to be chosen on runtime, through multiple dispatching. For instance, a `long` can be represented by a standard integer when it is small enough and by a big integer only when it is necessary.

3.4.2 Translation Toolchain

The translation toolchain consists of a pipeline of transformations, including:

- **Flow analysis** - each function is interpreted using a special objspace called *flow objspace*. This results in a flowgraph of linked objects, where each block has one or more operations;
- **Annotator** - the annotator assigns a type to the arguments, variables and results of each function;
- **RTyping** - the RTyping uses these annotations to expand high-level operations into low-level ones. For example, a generic `add` operation with operands annotated as integers will be expanded to an `int_add` operation;
- **Backend optimizations** - these include constant folding, store sinking, dead code removal, malloc removal and function inlining;
- **Garbage collector and exception transformation** - a garbage collector is added and exception handling is rewritten to use manual stack unwinding;
- **C source generation** - finally C code is generated from the low-level flowgraphs.

However, 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.

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.

The JIT and other features from the translation toolchain are not specific to Python. In fact, the interpreter and translation toolchain are designed to be independent from each other. This separation encourages the development of other dynamic languages, without the need to reimplement the features already present at the translation toolchain (most notably, the JIT).

3.5 PLT Spy

PLT Spy is an experimental Python implementation written in PLT Scheme (the former designation of Racket) and C, first released in 2003. It parses and compiles Python source-code into equivalent PLT Scheme code.[23]

PLT Spy's runtime library is written in C and linked to 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.

3.6 Other Experimental Implementations

Following the same philosophy, there are other experimental Python implementations which target pre-existing languages or virtual machines to allow integration between Python code and the target language's code.

Such implementations include:

- **RubyPython**, written in Ruby and targeting the Ruby Virtual Machine
- **Brython** and **Skulpt**, written in JavaScript
- **CLPython**, written in Common Lisp

As stated above, these are experimental implementations and currently they only support subsets of Python.

3.7 Comparison

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

	Language(s) written	Platform(s) targetted	Slowdown (vs CPython)	Std. library support
CPython	C	CPython's VM	1×	Full
Jython	Java	JVM	∼ 1×	Most
IronPython	C#	CLI	∼ 0.55×	Most
PyPy	RPython	C, JVM, CLI	∼ 0.16×	Most
PLT Spy	Scheme, C	Scheme	∼ 1000×	Full

Table 1. Comparison between implementations

Each implementation typically targets the same language/platform it was written in, usually because they were designed with the goal of allowing Python developers to integrate that platform's libraries in their programs. The main exception is PyPy, which was designed for speed and not integration.

In terms of performance, most of these implementations are on a par with CPython, except for PLT Spy, manly due to the bottleneck that constitutes the repeated conversion of data from Scheme's internal representation to CPython's internal representation, that is needed in order to use the Python/C API.

On the other hand, the use of the Python/C API allows PLT Spy to effortlessly support all of Python's standard library and third-party C-based extension modules.

In the next section we present our initial approach to an implementation of Python for DrRacket that provides most of the advantages of PLT Spy but overcoming the performance problem.

4 Solution

In order to achieve the objectives stated in section 2, we plan to implement a source-to-source compiler from Python to Racket.

The 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. Syntax-objects are Racket's built-in structures which associate s-expressions with its source location information (line number, column position, etc.) and lexical context.

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.

Phase 2 is automatically performed by Racket, therefore the rest of section will refer exclusively to phase 1, i.e. compiling Python source-code to Racket source-code.

4.1 General Architecture

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

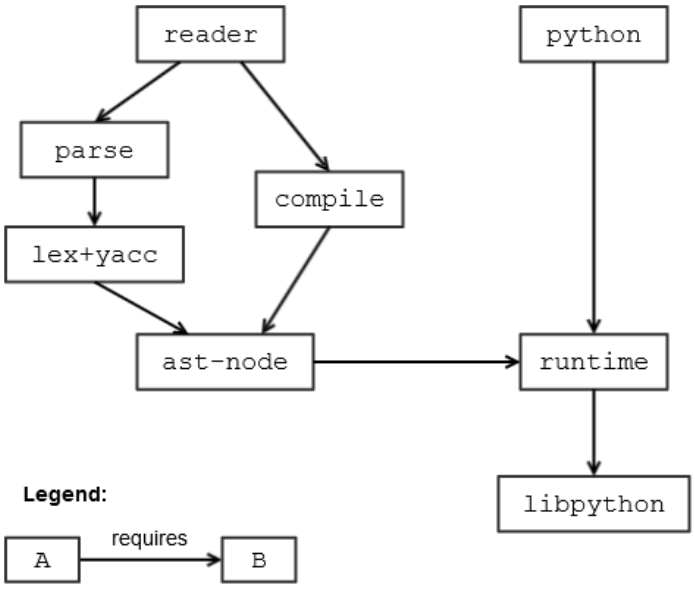

Fig. 2. Dependencies betwen modules. The arrows indicate that a module uses functionality that is defined on the module it points to.

4.1.1 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`). The entry-point for a `#lang` is at the `reader` module, visible at the top of **Fig. 2**. 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.

The `read` function is similar to `read-syntax`, but it simply takes an input port as the argument and returns a list of s-expressions instead of a list of syntax objects. It is usually coded by mapping the Racket function `syntax->datum` to the result of `read-syntax`, i.e. extracting the datum from each syntax object produced by `read-syntax`.

4.1.2 Parse and Compile Modules

The `lex+yacc` module defines a set of Lex and Yacc rules for parsing Python code. 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 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.

4.1.3 Runtime Modules

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. They are implemented on top of the actual CPython primitives, using the Python/C API through the Racket Foreign Function Interface (FFI). These FFI bindings are defined on the `libpython` module.

Finally, the `python` module simply provides everything defined at the `runtime` module, along with all the functionality from the `racket/base` language.

4.2 Parsing

There are two main alternatives for parsing Python code:

- Using Racket's parser tools to implement a lexer and parser for Python's grammar;
- Using Python's own parsing features, through its `ast` library. The resulting AST would be marshalled into a format such as XML or JSON and finally read and rebuilt in Racket.

The second alternative would yield an AST with an absolute guarantee of correctness (from the Python side), however the marshalling and unmarshalling

processes would take some time to implement and would still be error-prone, defeating the purpose of this method.

Therefore, we opted for the first alternative. The lexer turns the source-code into a sequence of tokens, which are then consumed by a LALR(1) parser. This task was as trivial as porting PLT Spy's lexer and parser code from PLT Scheme to Racket. There have been some changes in Python's grammar since version 2.3 (PLT Spy's Python version), but these are minimal and can be updated with little effort.

4.3 Code Generation

In order to support C-based Python modules, 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. 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 FFI to directly access the Python/C API in Racket, therefore our runtime is implemented in Racket.

While this provides us with a functional proof-of-concept, we will show on the Performance section that the repetitive use of these foreign functions introduces a huge overhead on our primitive operators, resulting in a very slow implementation.

We plan on improving it by exploring more efficient processes for implementing these primitive functions on top of Racket data types, therefore reducing our dependency on the FFI. This will only entail changing the `runtime` module, leaving the rest of the compilation process untouched.

4.4 Python 2 vs. 3

Before moving on to the examples, it is worth mentioning that Python's most recent version is Python 3.4, but we started our implementation effort by targeting version 2.7 (the final release of the 2.x series) for the following reasons:

- Python 3.x is intentionally not backwards compatible with Python 2.x and there are still many third party libraries not yet ported to 3.x[24];
- Python 2.x is still the default version for most current Linux distributions and for MacOS;
- Some of the new features in Python 3.x may be less intuitive for inexperienced programmers that are the main target audience for Rosetta. For instance, in Python 3.x the built-in function `range` now returns an iterator instead of a list.

It should be noted that this decision does not present a future upgrade to support the 3.x series as this version becomes more widely used.

4.5 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.

4.5.1 Ackermann

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

```
1  (define (ackermann m n)
2    (cond
3      [(= m 0) (+ n 1)]
4      [(and (> m 0) (= n 0)) (ackermann (- m 1) 1)]
5      [else (ackermann (- m 1) (ackermann m (- n 1)))]))
6
7  (ackermann 3 9)
```

Its equivalent in Python would be:

```
1  def ackermann(m,n):
2      if m == 0: return n+1
3      elif m > 0 and n == 0: return ackermann(m-1,1)
4      else: return ackermann(m-1, ackermann(m,n-1))
5
6  print ackermann(3,9)
```

Currently, this code is compiled to:

```
1  (define (:ackermann :m :n)
2    (cond
3      [(py-truth (py-eq :m #<cpointer>))
4       (py-add :n #<cpointer>)]
5      [(py-truth (py-and (py-gt :m #<cpointer>)
6                         (py-eq :n #<cpointer>)))
7       (py-call :ackermann (py-sub :m #<cpointer>) #<cpointer>)]
8      [else
9       (py-call
10       :ackermann
11       (py-sub :m #<cpointer>)
12       (py-call :ackermann :m (py-sub :n #<cpointer>)))]))
13
14 (py-print (py-call :ackermann #<cpointer> #<cpointer>))
```

Looking at line 1, 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`.

Please note that we still only support positional arguments on function definitions and calls. Python supports calling functions with an arbitrary number of keyword arguments. Those arguments that do not match the parameters declared on the function definition are placed on a special dictionary which is available inside the function's scope.

These semantics are not exactly the same as Racket's function calling semantics, but they can be emulated by compiling a function's body as a lambda and wrapping it with a higher-order function which would be responsible for binding the appropriate values to each parameter.

Starting at line 3, there are references to `#<cpointer>` values, this is due to the literals being evaluated and converted to CPython's internal representation at compile-time, therefore being displayed as pointers in the FFI's external representation.

The functions `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 are all treated as false when used on the condition of an `if`, `for` or `while` statement. All other values are treated as true.

The functions `py-call` and `py-print` implement the semantics of function calling and the print statement, respectively.

As a final remark, notice that except for the added verboseness, the original Racket code and the compiled code are essentially the same.

4.5.2 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 *iterations*.

```
1 (define (mandelbrot iterations c)
2   (let loop ([i 0] [z 0+0i])
3     (cond
4       [(> i iterations) i]
5       [(> (magnitude z) 2) i]
6       [else (loop (add1 i)
7                   (+ (* z z) c))])))
8
9 (mandelbrot 1000000 .2+.3i)
```

Its Python equivalent could be implemented like such:

```
1  def mandelbrot(iterations, c):
2      z = 0+0j
3      for i in range(iterations+1):
4          if abs(z) > 2:
5              return i
6          z = z*z + c
7      return i+1
8
9  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  (define (:mandelbrot :iterations :c)
2    (let ([:i (void)]
3          [:z (void)])
4      (let/ec return6306
5        (set! :z (py-add #<cpointer> #<cpointer>))
6        (py-for continue6305
7         (:i (py-call :range (py-add :iterations #<cpointer>)))
8         (begin
9           (cond
10            [(py-truth (py-or (py-gt :i :iterations)
11                              (py-gt (py-call :abs :z) #<cpointer>)))
12             (return6306 :i)]
13            [else py-none])
14          (set! :z (py-add (py-mul :z :z) :c))))
15       (return6306 :i))))
16
17 (py-print
18   (py-call :mandelbrot #<cpointer> (py-add #<cpointer> #<cpointer>)))
```

You will notice the `let` form on lines 2-3. 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.

A (more efficient) alternative would be compiling the first assignment of a given variable with a `let` form and the remaining assignments with `set!` forms within the enclosing `let`. This, however, would not guarantee the same scope semantics as Python's.

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 assigned in a compound statement is also visible outside of it.

Consider the case of variable `i`, first assigned inside a `for` loop (line 3 of the Python program). If its `let` form was placed inside the `for` loop, its enclosing scope would necessarily end when closing the `for` loop, therefore `i` would not be visible afterwards and the return statement on line 7 would yield an error.

Early returns are implemented as escape continuations, as seen at line 4: there is a `let/ec` form (syntactic sugar for `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 12.

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.

4.6 Performance

The charts on **Fig. 3** compare the running time of these examples for (a) Racket code on Racket, (b) Python code on CPython and (c) compiled Python code on Racket. These benchmarks were performed on an Intel® Core™ i7 processor at 3.2GHz running under Windows 7.

The times below represent an average of 3 samples.

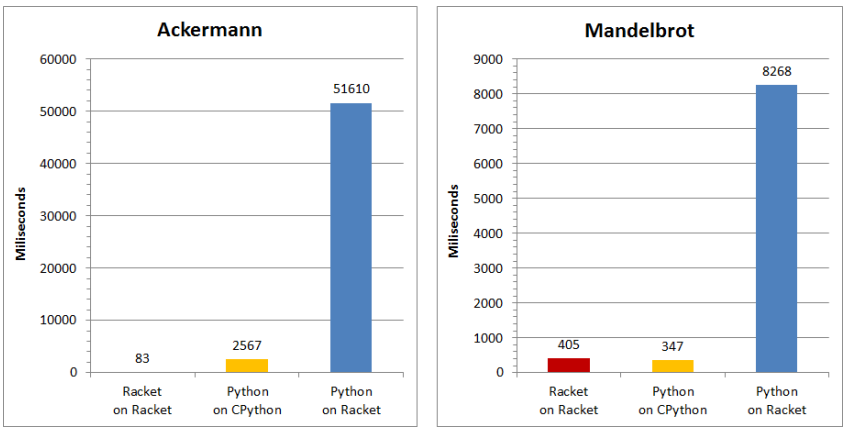

Fig. 3. Benchmarks of the Ackermann and Mandelbrot examples

We can see that Python code running on Racket is currently about 20-24 times slower than the Python code running on CPython, even for the Ackermann example, where the Racket implementation is significantly faster than Python's.

It is worth mentioning that the FFI function calls are allocating objects on the CPython virtual machine which are not being garbage collected, therefore producing a memory leak.

The Python/C API offers the same mechanism as the one CPython uses for garbage collecting (reference counting). This is meant to be used manually by the programmer: whenever a reference to a Python object is no longer needed, its reference count should be decremented; when it reaches zero, the object is garbage collected.

We are trying to implement this mechanism automatically in Racket by attaching a finalizer to each FFI function that allocates a new Python object. This finalizer is responsible for decrementing the object's reference count when Racket's GC proves that there are no more live references to the Python object.

Performance-wise, the use of the finalizers severely penalizes execution times: the Mandelbrot example now runs with an average time of $20384ms$ instead of $8268ms$, 18.5% of which are spent on garbage collecting. More work is needed to understand the causes of this performance loss.

5 Evaluation

Our implementation will be evaluated in two distinct aspects: correctness and performance.

The correctness of our implementation will be evaluated based on the formal specifications presented on [25] and [26] and by its conformance to their test suites. There are also test suites written for other Python implementations, therefore we will be using these as well.

Performance will be evaluated through benchmarks on several aspects of the Python language, comparing the running time and memory footprint of our implementation with the reference implementation, CPython.

Additionally, we will be evaluate the correctness and expressiveness of Rosetta's geometric modelling operators when using the Python front-end.

6 Conclusions

We have shown that there is a need for an implementation of Python for the Rosetta community and, more specifically, for DrRacket users. This implementation must support the most used Python libraries and should be competitive with 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.

Some of Python's common expressions and control flow statements have been already implemented, allowing for the successful compilation of two examples: the Ackermann function and a function for computing Mandelbrot's set.

In order to support CPython-specific libraries, we developed a runtime library based on CPython's primitive operations via the Racket Foreign Function Interface. This resulted in a proof-of-concept which already supports some of Python's features correctly, but suffers from poor performance and garbage collecting issues.

In the future, we will be exploring ways to implement Python's primitive operations over Racket types, therefore improving their performance and reducing the garbage collecting issues described at the end of section 4. We will also be implementing the remaining Python features.

References

1. Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. In *Journal of functional programming*, 12(2), 159-182.
2. Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S., & Felleisen, M. (1997, January). DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs* (pp. 369-388). Springer Berlin Heidelberg.
3. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., & Felleisen, M. (2011, June). Languages as libraries. In *ACM SIGPLAN Notices* (Vol. 46, No. 6, pp. 132-141). ACM.
4. Flatt, M., & Findler, R. B. (2013). *The Racket Guide*.
5. Lopes, J., & Leitão, A. (2011). Portable Generative Design for CAD Applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture* (pp. 196-203).
6. van Rossum, G., & Drake Jr, F. L. (1995). *Extending and embedding the Python interpreter*. Department of Algorithmics and Architecture, CWI.
7. Tröger, P. (2008, April). Python (2.5) Virtual Machine: A guided tour. `http://www.troeger.eu/files/teaching/pythonvm08.pdf`
8. VanRossum, G., & Drake, F. L. (2010). *The Python Language Reference*. Python Software Foundation.
9. Van Rossum, G. (2003). *An introduction to Python*. F. L. Drake (Ed.). Bristol: Network Theory Ltd..
10. Van Rossum, G., & Drake Jr, F. L. (2002). Python/C API reference manual. Chapter *Thread State and the Global Interpreter Lock*. Python Software Foundation.
11. Beazley, D. (2010, February). Understanding the python gil. In *PyCON Python Conference*. Atlanta, Georgia.
12. Beazley, D. M. (2009). *Python: essential reference*. Addison-Wesley Professional.
13. Differences between CPython and Jython. `http://jython.sourceforge.net/archive/21/docs/differences.html` (retrieved on December 2013).
14. Richthofer, S. (August 2013) JyNI - Using native CPython-Extensions in Jython In *EuroSciPi 2013*. Brussels, Belgium.
15. Juneau, J., Baker, J., Wierzbicki, F., Muñoz, L. S., & Ng, V. (2010). *The definitive guide to Jython: Python for the Java platform*. Apress.

16. Hugunin, J. (2004). *Dynamic Languages and the CLI: IronPython*. CLR Team, Microsoft.
17. IronPython .NET Integration documentation. `http://ironpython.net/documentation/` (retrieved on January 2014).
18. Ironclad - Resolver Systems `http://www.resolversystems.com/products/ironclad/` (retrieved on January 2014).
19. Hugunin, J. (2004). IronPython: A fast Python implementation for .NET and Mono. *PyCon. Python Software Foundation*, March.
20. PyPy Speed Center. `http://speed.pypy.org/` (retrieved on December 2013).
21. PyPy Compatibility. `http://pypy.org/compat.html` (retrieved on December 2013).
22. Brown, A., & Wilson, G. (2012). The Architecture of Open Source Applications, Volume II (Vol. 2). PyPy chapter (authored by Benjamin Peterson)
23. Meunier, P., & Silva, D. (2003, November). From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming* (pp. 24-29).
24. Python 2 or 3. `https://wiki.python.org/moin/Python2orPython3` (retrieved on December 2013).
25. Smeding, G. J. (2009). An executable operational semantics for Python. *Universiteit Utrecht*.
26. Guth, D. (2013). A formal semantics of Python 3.3. *University of Illinois at Urbana-Champaign*.