PyonR: A Python Implementation for Racket

[Extended Abstract]

Pedro Alexandre Henriques Palma Ramos
pedropramos@tecnico.ulisboa.pt

ABSTRACT

The Python programming language is becoming increasingly popular in a variety of areas, most notably among novice programmers, due to its readable syntax and extensive libraries. On the other hand, the Racket language and its DrRacket IDE have a tradition for being used to introduce Computer Science concepts to students. Besides, the Racket platform can be extended to support other programming languages. Both communities would benefit from an implementation of Python for Racket, since Racket programmers would be able to use libraries produced by the huge Python community and Python programmers would be able to access Racket's libraries and pedagogical tools, such as DrRacket.

This thesis proposes PyonR, an implementation of the Python language for the Racket platform. PyonR consists of a source-to-source compiler from Python to Racket and a runtime environment developed in Racket, which implements Python's language constructs and built-in functionality and enforces interoperability with Racket's data-types.

With this approach, we were able to implement Python's semantics with a very reasonable performance (on the same order of magnitude as other state-of-the-art implementations), full access to Python's libraries, a native interoperability between Racket and Python, and a good integration with DrRacket's features for Python development.

1. INTRODUCTION

Nowadays, the success of a programming language is not only determined by its inherent qualities, but also by the libraries available for it. It is usual for developers to have to leave the comfort of their preferred language and development environment in order to benefit from a library which is only available for another language.

DrRacket (formerly known as DrScheme) is an IDE for the Racket programming language (a descendant of Scheme and, thus, a dialect of Lisp) with a strong emphasis on pedagogy [4][5, p. 3]. Additionally, Racket and DrRacket support the development and extension of other programming languages [16]. These languages can be designed to interoperate with Racket libraries, thereby forming an ecosystem of "Racket languages". This ecosystem gives Racket users the liberty to write programs that mix modules in different languages and paradigms, therefore unifying the availability of libraries among different languages. Additionally, it gives DrRacket users the comfort of being able to integrate files in different languages within one single IDE.

The Racket language and DrRacket IDE have a tradition of being used to introduce Computer Science concepts in introductory programming courses, but lately, the Python language has been replacing Racket in many computer science courses.

Due to its large standard library, expressive syntax and focus on code readability, Python is becoming an increasingly popular programming language in many areas. If we consider the number of repositories created on GitHub in the last year (from October 2013 to September 2014) as a rough measure of a programming language's popularity, Python ranks in 5th place with around 214,000 repositories. Racket, on the other hand, only accounts for around 1,200 repositories. This suggests that a Python implementation for Racket with the ability to access Python code from Racket and vice-versa would be useful for both communities. On one hand, it would be beneficial for the Racket community to be able to access Python's countless libraries from Racket or being able to write programs that effortlessly mix Racket and Python code. On the other hand, it would be beneficial for Python programmers to be able to take advantage of Racket libraries and Racket tools such as DrRacket.

In fact, there is already a practical application for this implementation in Rosetta, an IDE based on DrRacket but specifically meant for generative design: an architectural design method based on a programming approach. Generative design allows architects to design complex three-dimensional structures that can then be effortlessly modified through simple changes in a program's code or parameters.

Rosetta's 3D modelling primitives and the communication with computer-aided design (CAD) applications is implemented in Racket, and therefore Rosetta is provided as a Racket library. Rosetta has been used extensively with the

Racket language for teaching programming and generative design to architecture students, but since the Racket language is generally unknown to architects who have not been exposed to this curriculum, the majority of the generative design community is not willing to use Rosetta with Racket.

In order to push Rosetta from a purely academic environment to an industrial environment, Rosetta has started supporting other programming languages. Python has been receiving a big focus in the CAD community, particularly after it has been made available as scripting language for applications such as Rhino or Blender. This justifies the need for implementing Python as another front-end language for Rosetta, i.e. implementing Python in Racket.

1.1 Goals

We propose PyonR (pronounced "Pioneer"), an implementation of the Python programming language for the Racket platform, which fulfills the following goals:

- **Correctness and completeness** – we implemented Python's language constructs, as well as its built-in types and operations, and the most commonly used parts of its standard library. Additionally, we provide access to third party libraries written for Python, including those written in C or other languages that compile to native code.

- **Performance** – our goal was not to produce the fastest Python implementation. Nonetheless, we achieved an acceptable performance on par with other state-of-the-art implementations.

- **Integration with DrRacket** – since DrRacket is the primary IDE for Racket development, we adapted its features in order to also provide a comfortable and productive user experience for Python development. These include the syntax checker and highlighter, debugger, REPL, among others.

- **Interoperability with Racket** – finally, we support the ability to import Racket libraries into Python code and vice-versa. The former is crucial in order to access Rosetta's features, which are provided by a Racket library. The latter introduces Python to the Racket language ecosystem, enabling Racket and its dialects (Typed Racket, Lazy Racket) to import functionality from Python libraries and files.

We assume that the reader is at least familiar with the basics of the Python language and also with the Racket language (or a similar dialect of Lisp) and its use of hygienic macros. More advanced features in either language will be conveniently explained when necessary.

In section 2, we explore some related state-of-the-art Python implementations. Sections 3-6 describe our solution in its different conceptual parts. Section 7 presents some performance benchmarks for PyonR. Finally, section 8 presents our conclusions.

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 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 from an interactive REPL) 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 [17].

2.2 Jython and IronPython

Jython is an alternative Python implementation, written by Jim Hugunin 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 module extensions written for CPython, but they can import Java classes, using the same syntax that is used for importing Python modules.

IronPython is another alternative implementation of Python, also developed by Jim Hugunin, but 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. Just like Jython, IronPython provides support for importing .NET libraries and using them with Python code [12].

In terms of speed, Jython claims to be approximately as fast as CPython. IronPython claims to be 1.8 times faster than CPython on `pystone`, a Python benchmark for showcasing Python's features.

Neither Jython nor IronPython support the Python/C API and therefore lack access to CPython's module extensions. This includes the great majority of the Python standard library, which had to be reimplemented, but it also includes some popular C-based libraries, such as NumPy (a library for high-performance arrays). There are, however, efforts by third parties to achieve this on both implementations: Ironclad [7] for IronPython and JyNI [9] for Jython.

2.3 PyPy

PyPy is yet another Python implementation, developed by Armin Rigo et al. and written in RPython, a heavily restricted subset of Python, in order to allow static inference of types. 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 [14].

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 [13].

Like the implementations mentioned before, an 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 JVM and CLI are also supported).

However, what truly makes PyPy stand out as currently the fastest Python implementation is its just-in-time (JIT) compiler, which detects common codepaths at runtime and compiles them to machine code, optimizing their speed. The JIT compiler 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 compiler works better for programs without frequent changes in loop conditions.

2.4 CLPython

CLPython (not to be confused with CPython, described above) is yet another Python implementation, developed by Willem Broekema and written in Common Lisp. Its development was first started in 2006, but stopped in 2013. 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 [2].

CLPython compiles Python source-code to Common Lisp code. Python objects are represented by equivalent Common Lisp values, whenever possible, or CLOS instances otherwise. Unfortunately, CLPython does not provide support for C module extensions, since it does not implement the Python/C API [3].

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.

2.5 PLT Spy

PLT Spy is an experimental Python implementation, developed by Daniel Silva and Philippe Meunier. It is written in PLT Scheme (Racket's predecessor) and C and was first released in 2003. It parses and compiles Python source-code into equivalent PLT Scheme code [11].

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 tradeoff in portability, 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.

	Platform(s) targeted	**Speedup (vs. CPython)**	**Std. library support**
CPython	CPython's VM	1×	Full
Jython	JVM	∼ 1×	Most
IronPython	CLI	∼ 1.8×	Most
PyPy	C, JVM, CLI	∼ 6×	Most
CLPython	Common Lisp	∼ 0.5×	Most
PLT Spy	PLT Scheme	∼ 0.001×	Full

Table 1: Comparison between implementations

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 libraries, 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.

PyPy is by far the fastest Python implementation, mainly due to its smart JIT compiler. However, 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, IronPython, and CLPython 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 or, alternatively, one must develop a way to access the Python/C API.

With these ideas in mind, we will be presenting our own solution in the next sections.

3. RUNTIME ENVIRONMENT

A Racket file usually starts with the line `#lang <language>` to specify in which of the available languages the file is written in. The Python language implemented by PyonR is specified with `#lang python` (this does not compromise portability across different Python implementations because the hash character starts a line comment in Python).

In order to implement a new language for the Racket platform, Racket requires two modules: (1) a reader module, which defines how the language's syntax is translated to Racket code, and (2) a language module, which acts as a runtime environment, i.e. a library that defines the functionality provided by the language. Our proposed solution, there-

fore, consists of (1) a source-to-source compiler which compiles Python code to semantically equivalent Racket code and (2) a runtime environment which provides not only the Racket library but also a set of functions and macros that define Python's primitive operations and its standard library.

This section describes the implementation of the runtime environment for PyonR. The compilation process will be described in section 4. In order to agree on a common terminology and make the tradeoffs of our decisions clear, let us start by briefly going over Python's data model.

3.1 Python's Data Model

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

The language's operator behaviour for each object is defined in its type-object's dict, as a method. For instance, the expression `a + b` (adding objects `a` and `b`) is roughly equivalent to `type(a).__add__(a,b)`. Therefore, the behaviour of the plus operator is determined at runtime, by computing `a`'s type-object and looking up the method mapped by the string `"__add__"` in its hash-table and its supertypes' hash-tables until it is found.

Besides additions, this behaviour is shared by all unary and binary operators, for getting/setting an attribute/index/slice, for printing objects, for obtaining their length, etc. [18]. For user-defined types, these methods can be defined during class creation (a `class` statement defines a new type-object), but they may also be changed dynamically at runtime. This flexibility in Python allows objects to change behaviour during the execution of a program, simply by adding, modifying or deleting entries from these hash tables, but it also forces an interpreter to constantly lookup these methods, contributing to Python's slow performance when compared to other languages.

3.2 Runtime Implementation Strategy

Taking into consideration the main ideas from each of the Python implementations described in section 2, we tried two alternative implementations:

The first one, inspired by PLT Spy, relied on using a foreign function interface [1] to map Python's operations into foreign calls to the Python/C API [19]. This allowed us to access every library supported by CPython, but, on the other hand, it suffered from two problems: (1) simple operations need to perform a significant number of foreign calls, which led to an unacceptably slow performance and (2) Python values would have to be explicitly converted to their Racket representation when mixing Python and Racket code, resulting in a clumsy interoperability.

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

Later, by reusing some of the features developed for the first approach, we were successful in developing a mechanism for importing Python libraries from CPython's virtual machine to the Racket-based data model from our second approach (described in section 5.3). This way, we are able to get the best of both worlds with the second approach, by keeping the enhanced performance and native Racket-Python interoperability obtained from reimplementing Python's runtime behaviour in Racket, while still being able to universally access every library available for CPython.

3.3 Optimizations

In order to take advantage of Racket's data model, we have implemented some general optimizations worth mentioning:

- **Early dispatch** – users can overload Python's arithmetic operators class definitions to add custom behaviour for a specific type. Nonetheless, a typical Python program will still rely on these operators mostly for handling numbers. Therefore, each operator implements an early dispatch mechanism for the most typical argument types, which skips the heavier dispatching mechanism described in Python data model's section. For instance, our implementation of the plus operator first tests if the arguments are both numbers (dispatching Racket's `+` function on success), otherwise if they are both strings (dispatching Racket's `string-append` function on success) and only then it will look for an `__add__` method.

- **Sequence iteration** – Python's iteration constructs (`for` statements, list comprehensions, among others) all rely on getting an iterator object from the sequence about to be iterated. This iterator has a `next` method which returns the next element or raises an exception signalling its exhaustion. In order to avoid this expensive procedure for built-in sequence types, we defined a `->sequence` function which returns an equivalent Racket sequence, which leads to a faster iteration.

- **Attribute getting and setting** – Python's attribute getting and setting operations rely on a sophisticated procedure involving descriptors (objects with `__get__` and `__set__` methods, that act as a proxy for an attribute). In order to hasten this procedure, we cache the `__get__` and `__set__` methods in the structure implementing our type-objects and we replicated these operations into two versions, one for standard objects and another for objects with an instance dictionary (e.g. objects defined by classes, module objects), therefore simplifying the logic for each version.

The performance gains from each of these optimizations will be later showcased on section 7.

4. COMPILATION

This section describes the compilation process. This includes the reader module and the macros on the language modules (since these macros will be used for code generation).

The following steps summarize the pipeline for compiling and interpreting Python code on PyonR:

1. **Lexical analysis** – Python source code is scanned into a sequence of tokens;
2. **Syntactic analysis** – these tokens are parsed into an abstract syntax tree (AST);
3. **Code generation** – the AST is traversed to generate semantically equivalent Racket code;
4. **Macro expansion** – the generated Racket code is syntax-checked by the Racket parser and its macros are recursively expanded;
5. **Racket bytecode compiler** – the resulting source-code is then fed to Racket's bytecode compiler which performs a series of optimizations (including constant propagation, constant folding, inlining, and dead-code removal) and produces Racket bytecode;
6. **Racket VM** – finally, this bytecode is interpreted on the Racket Virtual Machine, where it may be further optimized by a JIT compiler.

Steps (1) to (3) are performed by PyonR, while steps (4) to (6) are the Racket platform's responsibility. Therefore, the goal of our source-to-source compiler is to ensure that the generated Racket code is valid and semantically equivalent to the original Python source-code.

As an example, consider the trivial Python program which prints the result of $2 + 3$:

```
1 #lang python
2 print 2+3
```

- The lexical analysis phase produces a sequence of tokens: the `print` keyword, the number `2`, the plus operator, and the number `3`.
- The syntactic analysis phase produces an AST whose root is a print statement node, whose child is a binary addition expression, whose children are the literals `2` and `3`.
- The code generation phase traverses this AST in a top-down approach and produces the Racket code `(py-print (py-add 2 3))`. The procedures `py-print` and `py-add` are defined in the runtime environment, as they implement the semantics of Python's `print` statement and plus operator, respectively.

4.1 Lexical and Syntactic Analysis

We chose to implement our scanner and parser (i.e. lexical and syntactic analysers) by porting PLT Spy's scanner and parser (originally developed in PLT Scheme, for Python 2.3) to Racket and adding the new syntax features introduced until Python 2.7. This also allowed us to reuse its architecture and some of its functionality for code generation.

Therefore, PyonR's reader module includes a Lex specification [10] for producing a scanner and a Yacc specification [8] for producing an LALR(1) parser, both of which are implemented fully in Racket, using the `parser-tools` library). The grammar originally used by PLT Spy and ported to `parser-tools` is described at [15].

As just mentioned, the scanner and parser work together in a pipeline, taking Python source-code as input and outputting an AST. The AST nodes are implemented as Racket objects: there is a general superclass for the AST node and each subclass defines a particular type of statement or expression (e.g. `print` statement, binary addition expression). These subclasses define the particular fields needed for their children (e.g. a binary addition expression node will hold references to the left and right operands, which are both expression nodes).

4.2 Code Generation

Code generation in PyonR encompasses two processes: expanding each AST node into Racket source-code and macro expansion.

The AST superclass defines a virtual `to-racket` method, responsible for generating a syntax object with the compiled code and respective source location. Each AST node subclass overrides this method. A call to `to-racket` on the root of the AST works in a top-down recursive manner, as each node will eventually call `to-racket` on its children.

Syntax-objects 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 is crucial for taking advantage of most of DrRacket's features, as will be explained in section 6.

4.3 Source-code Translation

While our current approach effectively bridges the Python and Racket communities, allowing Racket programmers to access Python libraries, it would also be valuable for the Racket community to have an alternative approach based on software reengineering methods. This alternative would be based on a straight translation from Python code to readable Racket code. Although this deviates from the purpose of PyonR, it is still a relevant contribution, given the overall goals of this thesis work.

PyonR's compiled code depends on a runtime environment to guarantee Python's semantics, but we can also reuse our lexical and syntactic analysers and develop an additional

code generator to translate Python code to Racket code without dependencies.

This may be used to aid converting large Python libraries to human readable and maintainable Racket code. Since Python's and Racket's data models differ greatly, it is not feasible that a simple translator will be able to enforce the same semantics as the original Python code, but since the result of the translation is to be double-checked and debugged by programmers, the focus is not on correctness, but on readability.

Given the potential complexity of this task and the fact that it shifts from our original goal, we have only implemented a proof-of-concept which may serve as a basis for future work.

5. INTEROPERABILITY

PyonR would not be complete without supporting Python's `import` statements, used to import functionality from other Python files. Additionally, it would have a very limited practical use if one could not also import Racket modules, particularly Rosetta. Finally, we also implemented an approach for importing functionality from CPython's virtual machine, which is particularly useful for reaching Python's standard library or other C based libraries only available to CPython.

5.1 ...with Python source code

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 different syntaxes to import modules in Python

1. `"import" <module> ["as" <name>]` - makes <module> available as a new binding for a module object. The bindings defined inside that module are accessible as attributes. The optional "as" clause may be used to define some other name for the imported binding;
2. `"from" <module> "import" (<id> ["as" <name>])+` - the <id> bindings defined inside <module> are made available as new bindings. The module object is not made available to the user. Imported bindings may also be renamed with the "as" clause;
3. `"from" <module> "import" "*"` - similar to the above, but provides all bindings defined inside <module> using their original names.

For the first syntax (`import...as`), we make use of `module->exports` to get a list of the bindings provided by the module and `dynamic-require` to import the names which do not start by an underscore. Finally, we define the specified name as a new module object containing these imported bindings.

The second syntax (`from...import...as`) is simpler: we simply define the specified names as calls to `dynamic-require`.

The third syntax (`from...import*`) entails defining new bindings whose names are unknown until the import happens. These defined names must be known at macro-expansion time, so that they can be defined and further references to them will not be signalled as syntax errors for unbound identifiers. Thus, the module must be located, declared, and visited at macro-expansion time, but only instantiated at runtime. To declare and visit the module, we use `namespace-require/expansion-time`. This will perform macro-expansion on the required module, which is enough for us to obtain the exported names with `module->exports`. This way, we can generate the `define` forms for each of the required names, whose bindings are obtained with `dynamic-require` at runtime. This effectively postpones module initialization (i.e. evaluating the module's code) to runtime, so that eventual side-effects take place according to a correct order.

5.2 ...with Racket

In order to import Racket modules, we chose to support a similar syntax to Python's import statements. The only difference is that instead of specifying a module's name, the user enters a `require` specification as a Python string literal (with a mandatory `"as"` clause for the first syntax).

This way, in addition to keeping the familiarity and expressiveness of Python's import statements, we also get all the expressive power of Racket's `require` forms. We can, therefore, import any module available to Racket, such as installed packages and collections, arbitrary files, collections from PLaneT: Racket's centralized package distribution system, etc.

```
#lang python
import "racket" as rkt
import '(file "/path/to/module/bar.rkt")' as bar
import "(planet aml/rosetta)" as rosetta
```

The first syntax is implemented similarly, but the second and third ones are now implemented using traditional `require` forms, since this time the module's location can be fully determined by its specification at compile-time. One neat side effect of this is that Racket macros can be imported and used to some extent.

Also, since the set of allowed identifier names in Python is a subset of those in Racket, we attempted to make most Racket identifiers reachable in Python through name mangling. We replace hyphens with underscores and other forbidden characters are replaced by an alphabetical representation separated by underscores. We also prefix names starting with a digit by an underscore. For instance, the identifier `3+2` (valid in Racket) would be imported to Python as `_3_PLUS_2`.

5.2.1 *Predicate dispatching*

Even though we are now able to import Racket libraries into Python, using these libraries with Python code may still feel like we are not taking advantage of Python's idioms. In Python, every value has a well-defined type and, typically, every part of its functionality is encapsulated inside that type. Python programmers can define new types with the `class` statement and define special methods which interact with the language's constructs. On the other hand, Racket values do not have a well-defined type. Instead, values are recognized by opaque predicates. For instance, the list `'(a`

`b c)` is recognized by the `list?` predicate, but also by the `pair?` predicate.

In order to make Racket libraries feel more "Pythonic", we have come up with an extensible predicate dispatch mechanism to associate Python types to Racket predicates. The user can add mappings $(pred, type)$ between predicates and type-objects, so that if a Racket value is recognized by *pred*, its Python type will be *type*.

Additionally, since predicates are opaque, there is no general way to automatically establish a hierarchy between them. Therefore, when there is more than one predicate recognizing an object, it is impossible to decide which one should be favoured.

By default, our implementation chooses the most recently entered predicate, but in order to allow users to specify more robust criteria, we also included a way to defines that a predicate *pred* is a subtype of predicate *parent_pred*, i.e. $pred(x) => parent\_pred(x)$.

Whenever the $(pred, type)$ association list is changed, it is stably sorted so that if *pred*1 is a subtype of *pred*2, then *pred*1 will show up before *pred*2. In addition, subtypes are transitive, i.e. if *pred*1 is defined as a subtype of *pred*2 and *pred*2 is defined as a subtype of *pred*3, then *pred*1 is a subtype of *pred*3. This means that, in the event of an overlapping set of predicates, our implementation will dispatch the most specific one.

5.2.2 *Importing Python from Racket*

So far we have described mechanisms to import Python and Racket functionality into Python source code. Since these are implemented in Racket itself, it is not surprising that they can also be used to import Python functionality into Racket source code.

We provide the three `import` syntaxes as Racket forms, so that Python objects can be imported and used in Racket. Built-in types are provided, generally with a `"py-"` prefixing their names. For instance, the `object`, `int` and `str` types are provided as `py-object`, `py-int` and `py-string` respectively.

Callable Python objects can be called directly in Racket since they all implement the `prop:procedure` property [6]. As for other Python operators, we provide the same Racket functions and macros we use on compiled code. These include the `py-get-attr` and `py-set-attr` functions for getting and setting attributes, the `py-add`, `py-sub`, `py-mul` and `py-div` as each one of Python's `+`, `-`, `*` and `/` operators, among others.

5.3 ...with CPython

As mentioned earlier, most of Python's huge standard library is not implemented in Python, but in the C programming language. Additionally, there are also numerous third-party libraries for Python which are fully or partially implemented in C. One very popular case is NumPy, a library widely used for scientific computing, whose main feature is a very fast implementation of N-dimensional array objects. Being able to take advantage of CPython's libraries with minimal performance overheads would therefore be extremely valuable to the Racket platform, as it would enable fast access to a variety of new libraries, including some with better performance than their Racket equivalents.

Importing a module directly from CPython entails a radically different approach from the `require` and `dynamic-require` methods described earlier. Therefore, we provide this feature as an additional import mechanism simply by replacing the `import` keyword with `cpyimport`.

The module objects are imported from the Python/C API using `PyImport_Import` [19]. Just like with our initial strategy for implementing a runtime environment, linking Racket with the Python/C API is made possible through the Racket Foreign Function Interface (FFI). This means that calls to the Python/C API return C pointers to CPython objects allocated in shared memory. The core aspect of this strategy relies on converting these C pointers to equivalent `#lang python` objects back and forth.

This is accomplished by two general-purpose functions:

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

Both functions start by figuring out the argument's type and then dispatch its conversion to a more specific function. Some types have a known structure (numbers, strings, type-objects, modules) and can therefore be converted to our Racket representations.

Objects whose internal representation we do not know (e.g. the types defined in the libraries we are importing), opaque objects (e.g. Python functions) and mutable objects which could be updated "behind our backs" (e.g., lists and dicts) are converted to a proxy-objects.

Proxy-objects are simple wrappers that contain a reference to the converted type of the object and its C pointer as returned by the FFI. They act as proxies, in Racket, for the Python objects in CPython's shared memory. Converting a proxy object back to its CPython representation is as simple as unwrapping the stored C pointer.

Like before, these `cpyimport` syntaxes are available as Racket forms, so that objects imported from CPython can be used with the Racket language.

6. INTEGRATION WITH DRRACKET

As the reader may recall, the proposed goals for this implementation include adapting DrRacket's features (mainly intended for the Racket language) to handle Python code. The DrRacket IDE is shipped with the Racket platform and a great part of it can be configured in Racket. Since PyonR is based on the Racket platform, it makes sense that a PyonR user should be able to take advantage of DrRacket for a comfortable, productive and error-free Python programming experience.

We have adapted the following features:

- **Source-code location** – during the lexical analysis phase, each generated token contains the location of the piece of source-code it refers to, i.e. the starting and ending line and column numbers. These source-code locations are kept during the syntactic analysis phase, being stored on each AST node instance. Finally, the syntax-objects produced by the code generation phase from each AST node are filled with these source-code locations. This way, DrRacket's features which rely on presenting info related to the source-code will work for Python. This includes displaying syntax errors, tracking a debugging session, and drawing an arrow from an identifier to the statement where it was bound.
- **Syntax highlighting** – to implement a new syntax highlighter, tailored for Python code, we reused some parts of the lexer used for compilation to implement a similar but simpler lexer, whose only purpose is to produce tokens containing values from Racket's color scheme and source-code location.
- **Read-Eval-Print-Loop** – we changed the way DrRacket's REPL should behave for the Python language when pressing Enter, in the decision to submit the code for evaluation or wait for more input. We use the lexical analysis lexer to transform the input into tokens and analyse them to determine if the statement is complete. We also changing the way DrRacket prints values, as to conform to Python's semantics.
- **Error display handler** – we adapted the error display handler to display unhandled exceptions accordingly, by converting the exception from our representation to a Racket user error and calling the default error display handler on it.

All of these changes are configured for `#lang python` only, therefore they do not interfere with the default behaviour for other Racket languages.

7. PERFORMANCE

In order to measure PyonR's current performance and the gains obtained from each one of the optimizations mentioned on section 3.3, we have tested 5 examples, each one stressing different features of the Python language. The code used for each one of these examples can be found at `https://github.com/pedropramos/PyonR/tree/master/examples`.

These benchmarks were performed on an Intel Core i7 processor at 3.2GHz, running Windows 7. The times mentioned below correspond to the minimum out of 5 samples.

7.1 Ackermann

Consider the implementation of the Ackermann function in both Python and Racket. It stresses the performance of recursive function calls and arithmetic operators on integers. The running times are shown on **Fig. 1**.

For this example, the Racket implementation runs incredibly faster than CPython's, mostly due to Racket's lighter function calls and operators, as the Ackermann example heavily depends on them. This is very beneficial for our implementation, because even though we had to implement Python's semantics, they are implemented on top of Racket functionality.

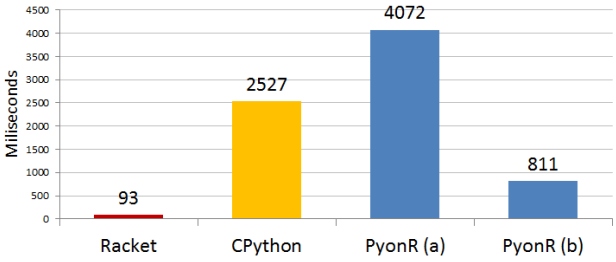

Figure 1: Benchmark of the Ackermann function

Likewise, even without early dispatch, this yields a very close performance to CPython. Optimizing the dispatching mechanism of operators for common types further led to huge gains in this example, pushing it below the running time for CPython to around 3 times faster.

7.2 Mandelbrot

Consider a Racket and Python programs which define and call 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. This example stresses the `for` loop and early returns (which are implemented with escape continuations). The running times are shown on **Fig. 2**.

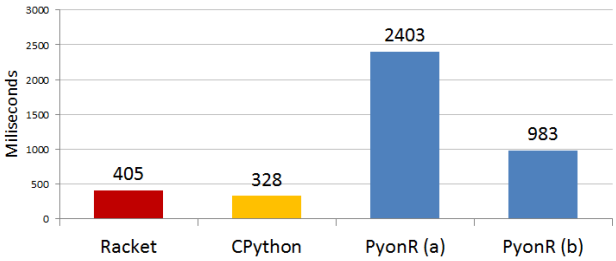

Figure 2: Benchmark of the Mandelbrot function

This time, the Racket implementation is slightly slower than CPython's. Still, PyonR does a good job at approaching CPython's time, taking advantage of the optimization on the way Python lists are now iterated by the `for` loop. In the end, PyonR stand at around 3 times slower than CPython for this example, which is very good given that Python's semantics are implemented over Racket.

7.3 Mandelbrot using Classes

Consider now a variation of the last example where we store the `z` variable as the attribute of an object. This example stresses attribute getting and setting.

Since Python classes do not have a direct mapping in Racket with similar semantics, we chose not to include a Racket implementation. Still, the chart on **Fig. 3** presents the running times for this example.

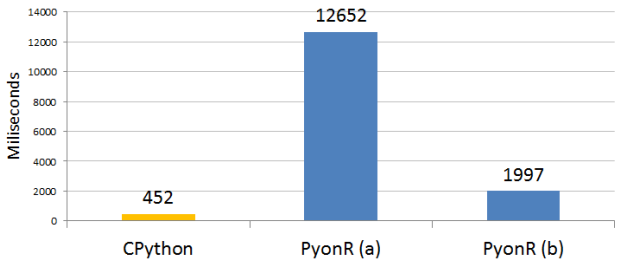

Figure 3: Benchmark of the Mandelbrot function using classes

This example demonstrates how significant our optimizations to the attribute getting and setting methods were, given their frequent use in a typical program with user-defined classes. Nonetheless, the end result shows that CPython's running time went up by around 38% from the previous example while PyonR's running time doubled. This suggests that attribute getting and setting may still be a bottleneck in PyonR.

7.4 Pystone

Pystone is a Python translation of the Dhrystone benchmark that is distributed with CPython. It is meant to be a general benchmark for single threaded programs, stressing the most common Python features.

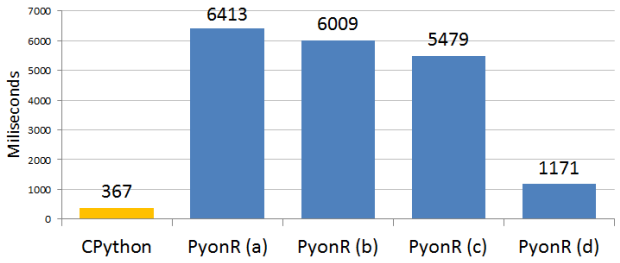

Figure 4: Pystone benchmark

The effects of early dispatch and sequence iteration optimization are not as visible on Pystone as they were on previous benchmarks, mostly because this benchmark does not have such a heavy dependency on arithmetic operators and `for` loops. Nonetheless, this serves to confirm that these optimizations still have a positive effect on more generic programs.

On the other hand, the effect of the optimizations on attribute getting and setting is as noticeable as it was on the prior benchmark. The end result shows that PyonR is currently about 3 times slower than CPython on Pystone, which seems to generally support some of the previous benchmarks.

8. CONCLUSIONS

A Racket implementation of Python would benefit Rosetta users, allowing them to take advantage of Rosetta's generative design features using the Python programming language, but also Racket developers in general, giving them access to Python's huge standard library and the ever-growing universe of third-party libraries, and Python developers, by providing them with a pedagogic IDE in DrRacket. In order to be relevant for this target audience, 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. It should also be able to take advantage of DrRacket's features.

Our solution, PyonR, consists of a source-to-source Python-to-Racket compiler and a runtime environment which implements Python's constructs and standard library over Racket functionality.

We had proposed goals in 4 different levels, which we address here:

- **Correctness and completeness** – PyonR currently implements most of Python 2 (i.e. every language construct except for the `del` and `with` statements, decorators, and the `super` function for constructors), which allows for the successful compilation and execution of a very large subset of Python programs. We have implemented a very small subset of Python's standard library in Racket, but nonetheless PyonR users can access every module provided by the standard library, as well as any external Python libraries, with the `cpy-import` statement (section 5.3), provided that these are installed on CPython, Python's reference implementation.
- **Performance** – PyonR's current performance varies between 3 times slower than CPython for the Pystone benchmark and 3 times faster for handling recursive function calls, as supported by the benchmarks described on section 7. These results are not surprising, considering that the time available for this implementation was roughly one year. We believe it is still possible to improve these results with some profiling effort.
- **Integration with DrRacket** – by following Racket's guidelines for language implementation and providing alternative implementations of DrRacket's tools, we were successful in reusing many of DrRacket's features such as syntax-highlighting, the read-eval-print-loop and the debugger when programming in Python. Additionally, by keeping track of the original Python source-code location during compilation, we are able to display arrows for lexical bindings, report syntax errors and display stack-traces for exceptions, as was illustrated in section 6.
- **Interoperability with Racket** – we extended the syntax on Python's `import` statement to be able to import Racket libraries without any type conversion or additional overhead, and we also developed a mechanism to integrate Racket values with Python's type system (section 5.2). This gives PyonR users the ability to use any Racket library from Python, including Rosetta, and even some limited support for Racket macros, given the restrictions imposed by Python's syntax. Importing Python libraries into other Racket languages is also possible by requiring the Python runtime system, for which we provide the Racket constructs which implement its functionality.

The source-code for PyonR is available on GitHub at `https://github.com/pedropramos/PyonR`. It requires Racket version 5.92 or above and Python version 2.7. Since both platforms are available for multiple operating systems, including Windows, Mac OS X, and Linux, PyonR is also available for multiple OSs. It has been tested on Windows 7 and Debian.

8.1 Rosetta

For the particular case of Rosetta, our main goal with this implementation was to add support for using Rosetta (and its supported back-end applications) with Python as the front-end programming language.

This goal was fulfilled in every way. By directly mapping some Python types to Racket equivalents (e.g. Python's numbers to Racket's numerical tower) and developing a modified `import` statement which maps to Racket `require` forms (section 5.2), we achieved a native interoperability with the Racket platform and its libraries, allowing us to support in Python every feature of Rosetta that was supported in Racket.

Likewise, Rosetta can now be imported from the Python language, using DrRacket, and its modelling primitives can be used as easily as they would in the Racket language.

8.2 Future Work

PyonR can already be used with Racket to write and run full Python programs, but future work includes implementing the few remaining Python language constructs and completing the implementation of its built-in type-objects with their remaining methods, so that the implementation's correctness can be verified through unit testing.

In terms of performance, there is still much that can be done, both at compilation and runtime, in order to speed up PyonR. Such optimizations could include rewriting the control flow for or functions with `break`, `yield` or early `return` statements, to avoid using escape continuations, compiling frequently used Python idioms to simplified Racket forms with the same semantics, rewriting parts of the runtime environment in Typed Racket to avoid the performance overhead of checking types at runtime, and profiling Python examples to find and optimize the implementation's weakpoints.

The source-code translation backend of the compiler (section 4.3) can be extended to support more complex Python constructs and provide a better accuracy in its compilation results, with techniques such as type inference, as mentioned.

Finally, if the interest arises, PyonR can be migrated to support Python 3 and follow its release schedule with new features.

References

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

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

[3] W. Broekema. *CLPython Manual*, chapter "10.6 Compatibility with CPython C extensions". 2011.

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

[5] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer Berlin Heidelberg, 1997.

[6] M. Flatt et al. *The Racket Reference*, chapter "4.17 Procedures". PLT, 2013.

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

[8] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

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

[10] M. E. Lesk and E. Schmidt. Lex: A lexical analyzer generator, 1975.

[11] 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.

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

[13] PyPy compatibility. `http://pypy.org/compat.html`. [Online; retrieved on October 2014].

[14] PyPy speed center. `http://speed.pypy.org/`. [Online; retrieved on October 2014].

[15] D. Silva. Implementing a Python to Scheme compiler. `http://plt-spy.sourceforge.net/papers/pts.pdf`, April 2004.

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

[17] G. van Rossum and F. L. Drake. *Extending and embedding the Python interpreter.* Centrum voor Wiskunde en Informatica, 1995.

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

[19] G. van Rossum and F. L. Drake Jr. *Python/C API Reference Manual*, 2002.