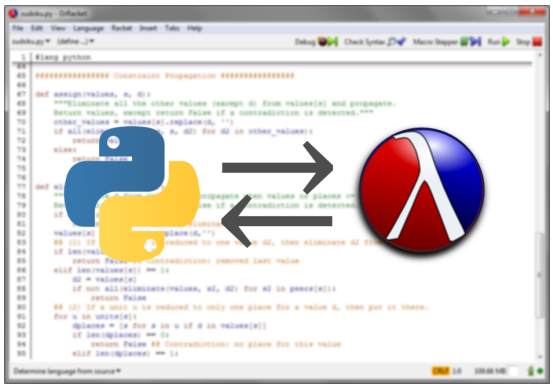
TÉCNICO LISBOA



PyonR: A Python Implementation for Racket

Pedro Alexandre Henriques Palma Ramos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson:	Prof. Dr. José Manuel da Costa Alves Marques
Supervisor:	Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member of the Committee:	Prof. Dr. João Coelho Garcia

October 2014

Agradecimentos

Agradeço...

Em primeiro lugar ao Prof. António Leitão, por me ter dado a oportunidade de participar no projecto Rosetta com esta tese de mestrado, por todos os sábios conselhos e pelos momentos de discussão e elucidação que se proporcionaram ao longo deste trabalho.

Aos meus pais excepcionais e à minha mana preferida, por me terem aturado e suportado ao longo destes quase 23 anos e sobretudo pelo incondicional apoio durante estes 5 anos de formação superior.

Ao pessoal do Grupo de Arquitectura e Computação (Hugo Correia, Sara Proença, Francisco Freire, Pedro Alfaiate, Bruno Ferreira, Guilherme Ferreira, Inês Caetano e Carmo Cardoso), por todas as sugestões e pelo inestimável feedback em artigos e apresentações.

Aos amigos em Tomar (Rodrigo Carrão, Hugo Matos, André Marques e Rui Santos) e em Lisboa (Diogo da Silva, Nuno Silva, Pedro Engana, Kaguedes, Clara Paiva e Odemira), por terem estado presentes, duma forma ou doutra, nos essenciais momentos de lazer.

À Fundação para a Ciência e Tecnologia (FCT) e ao INESC-ID pelo financiamento e acolhimento através da atribuição de uma bolsa de investigação no âmbito dos contratos Pest-OE/EEI/LA0021/2013 e PTDC/ATP-AQI/5224/2012.

Finalmente, ao Instituto Superior Técnico, pela possibilidade de ter conhecido pessoas extraordinárias e pela sua cultura de exigência, através da qual me habituei a procurar sempre o rigor e o prestígio em todas as minhas iniciativas.

Lisboa, 14 de Outubro de 2014

Pedro Palma Ramos

Resumo

A linguagem de programação Python tem ganho popularidade em várias áreas, sobretudo entre programadores principiantes, devido à sua sintaxe particularmente legível e bibliotecas diversas. Por outro lado, a linguagem Racket e o ambiente de desenvolvimento DrRacket têm a tradição de serem usados para introduzir conceitos de Informática a alunos. Além disso, a plataforma Racket oferece a possibilidade de ser alargada com outras linguagens de programação. Ambas as comunidades beneficiariam duma implementação de Python para Racket, pois, desta forma, os programadores de Racket poderiam usar bibliotecas produzidas pela enorme comunidade de Python e os programadores de Python poderiam aceder às bibliotecas e ferramentas pedagógicas do Racket, tais como o DrRacket.

Esta tese propõe o PyonR, uma implementação da linguagem Python para a plataforma Racket. O PyonR consiste num compilador source-to-source de Python para Racket e um ambiente de runtime desenvolvido em Racket, que implementa os elementos da linguagem Python e a funcionalidade incluída na linguagem e garante a interoperabilidade com os tipos de dados de Racket.

Com esta abordagem, conseguimos implementar a semântica da linguagem Python com uma performance muito razoável (na mesma ordem de grandeza que outras implementações do estado da arte), acesso total às bibliotecas de Python, uma interoperabilidade nativa entre Racket e Python e uma boa integração com as capacidades do DrRacket para a programação em Python.

Palavras-chave: Racket, Python, Interoperabilidade, Compiladores, Ambientes de Runtime

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.

Keywords: Racket, Python, Interoperability, Compilers, Runtime Environments

Contributions

Some of the contributions presented in this thesis have also been published in three other papers, namely:

- *An Implementation of Python for Racket*, published in the *7th European Lisp Symposium* [29]:
- *Implementing Python for DrRacket*, published in the *3rd Symposium on Languages, Applications and Technologies* [30];
- *Reaching Python from Racket*, published in the *8th International Lisp Conference* [31].

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CAD	Computer Aided Design
CLI	Common Language Infrastructure
CLOS	Common Lisp Object System
FFI	Foreign Function Interface
GC	Garbage Collector
GIL	Global Interpreter Lock
IDE	Integrated Development Environment
JIT	Just-In-Time
JVM	Java Virtual Machine
LALR	Look-Ahead LR
MRO	Method Resolution Order
REPL	Read-Eval-Print-Loop
VM	Virtual Machine

Chapter 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.

1.1 Racket and DrRacket

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 [8][9]. Unlike IDEs such as Eclipse or Microsoft Visual Studio, DrRacket provides a simple and straightforward interface particularly aimed at inexperienced programmers. It also provides a small set of useful productivity tools, including automatic syntax highlighting and syntax checking, a macro stepper, a debugger, and a profiler.

Additionally, Racket and DrRacket support the development and extension of other programming languages [35]. These languages can be designed to interoperate with Racket libraries, thereby forming an ecosystem of "Racket languages", in a similar fashion to the JVM languages (Java, Scala, Clojure, etc.) or the CLI languages (C#, Visual Basic, F#, etc.). The Racket ecosystem already includes implementations of some dialects of Racket (Typed Racket and Lazy Racket), but also other non-related languages (Datalog and Algol 60).

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.

1.2 Python

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. According to Peter Norvig [23], Python is an excellent language for pedagogical purposes and is easier to read than Lisp dialects for someone with no experience in either language.

Python is a high-level, interpreted, dynamically typed programming language [38, p. 3]. It supports the functional, imperative, and object-oriented programming paradigms and features automatic memory management.

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. Even if we combine Racket with other popular dialects of Lisp, namely Scheme, Common Lisp, Emacs Lisp, and Clojure, we get about 20,000 repositories which still falls short compared to Python.

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.

The Python language already has alternative implementations for the JVM (Jython) and the CLI (IronPython). Its reference implementation, CPython, is written in the C programming language and it is maintained by the Python Software Foundation. While most Python libraries are written in Python, some popular libraries are written in C (mainly for performance reasons), including most of Python's standard library. This means that, in order to provide universal access to Python libraries as intended, our implementation must also support a way to access native code.

1.3 Rosetta IDE

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 supports multiple back-ends for 3D visualization (including AutoCAD and Rhinoceros, two CAD applications). Users can effortlessly change from one CAD application to another by simply changing one line of code in their programs (**Fig. 1.1**).

Rosetta's 3D modelling primitives and CAD communication system is implemented in Racket, and

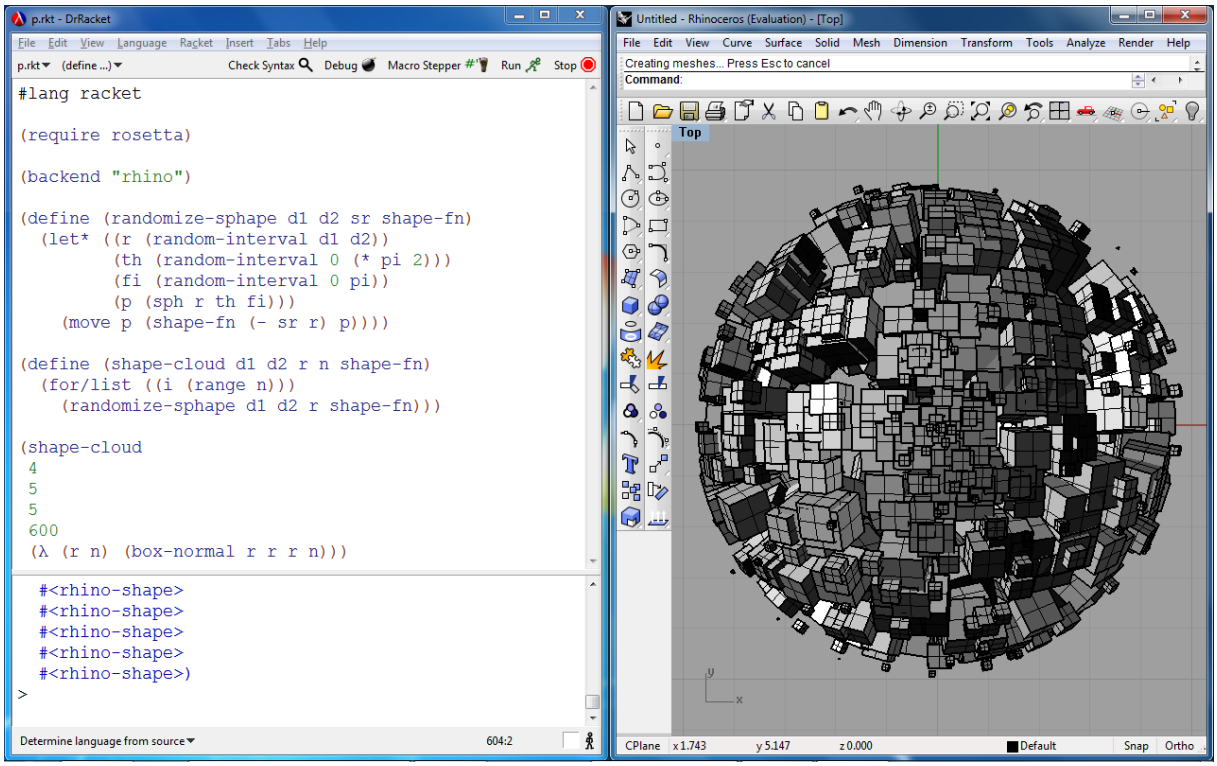


Figure 1.1: Rosetta being used with the Racket language as front-end and Rhinoceros as back-end. Within DrRacket, the language is selected with the `#lang` syntax. The backend can then be selected with the `backend` procedure, provided by Rosetta.

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. Currently, Rosetta supports front-ends for Racket, AutoLISP, JavaScript, and RosettaFlow (a graphical language inspired by Grasshopper). AutoLISP and JavaScript were chosen precisely because they have been traditionally used for generative design. More recently, 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.

1.4 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 (this would be very improbable considering that we are implementing over a very high-level language). 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.

In 2008, the Python Software Foundation introduced Python 3, which acted as a major revision to the Python language, breaking backwards compatibility with previous versions. This led to somewhat of a rift in the Python community as some users adopted Python 3, while others resisted the change and remained using Python 2. Python 2 is no longer being upgraded with new language features, but its final release (Python 2.7) is still being supported, with an end-of-life date set for 2020 [26].

We chose to target Python 2 instead of Python 3, mainly because most of the related work is based on Python 2 (version 2.7 or earlier) and because Python 2 is still arguably the most used version, as it is the one shipped with most current Linux distributions and Mac OS. Nonetheless, it should be noted that this decision does not prevent a future upgrade to support Python 3 as it becomes more widely used.

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 chapter 2, we will explore some related state-of-the-art Python implementations. Chapters 3-6 will describe our solution in its different conceptual parts. Chapter 7 will present some performance benchmarks for PyonR. Finally, chapter 8 will present our conclusions.

Chapter 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, started by Guido van Rossum and now maintained by the Python Software Foundation, is written in the C programming language and has been the reference implementation of Python since its first release. It parses Python source code (from `.py` files or 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 [37].

2.1.1 Object Representation

CPython's virtual machine is a simple stack machine, where the byte codes operate on a stack of `PyObject` pointers [36].

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

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

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

integers will point to the instance on the pool instead of allocating a new one.

2.1.2 Garbage collection and Threading

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

Reference counting, however, does not work well with reference cycles [39, ch. 3.1]. Consider the example of a list containing a reference to itself. When its last reference goes out of scope, its counter is decremented, however the circular reference inside the list is still present, so the reference counter will never reach zero and the list will not be garbage collected, even though it is already unreachable.

Furthermore, these reference counters are not thread-safe [41]. If two threads would attempt to increment an object's reference counter simultaneously, it would be possible for this counter to be erroneously incremented only once. To avoid this from happening, CPython enforces a global interpreter lock (GIL), which prevents more than one thread running interpreted code at the same time.

This is a severe limitation to the performance of threads on CPU-intensive tasks. In fact, using threads will often yield a worse performance than using a sequential approach, even on a multiple processor environment [3]. Therefore, the use of threads is only recommended for I/O tasks [4, p. 444].

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

2.2 Jython 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.

Garbage collection in Jython is performed by the JVM and does not suffer from the issues with reference cycles that plague CPython [16, p. 57]. In terms of speed, Jython claims to be approximately as fast as CPython. Some libraries are known to be slower because they are currently implemented in Python instead of Java (in CPython these are written in C). Jython's performance is also deeply tied to performance gains in the Java Virtual Machine.

IronPython is another alternative implementation of Python, 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 [22].

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 [13].

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) and SciPy (a package of algorithms and mathematical tools widely used by the scientific computing community). There are, however, efforts by third parties to achieve this on both implementations.

2.2.1 Ironclad

Ironclad is an open-source project developed by William Reade since 2008 and supported by Resolver Systems [14], whose goal is to make Python C module extensions available to IronPython, most notably NumPy and SciPy.

Ironclad tries to achieve this by replacing the library implementing the Python/C API with a stub which intercepts Python/C API calls and impersonates them using IronPython objects instead of the usual CPython objects. For objects whose types are defined in a compiled C module extension, they have an IronPython type which wraps around them and forwards all method calls to the real Python/C API.

NumPy and SciPy already work with Ironclad. No benchmarks are provided, however the author mentions that performance is generally poor compared to CPython. He claims that "in many places it's only a matter of a few errant microseconds (...) but in pathological cases it's worse by many orders of magnitude" [7].

2.2.2 JyNI

JyNI is another compatibility layer, being developed by Stefan Richthofer since 2013 [17], whose goal is similar to Ironclad's but it's meant for Jython instead of IronPython.

It is still in an early phase of development (alpha) and does not yet support NumPy, but it already supports some of Python's built-in types. It uses a mix of three strategies for bridging objects from CPython to Jython and vice-versa [32]:

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

2. For the types where the Python/C API uses preprocessor macros, objects created on the CPython side are mirrored on the Jython side. For immutable objects this is trivial because there is no

need for further synchronization. Mutable objects are mirrored with Java interfaces which provide access to the object's shared memory;

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

2.3 PyPy

PyPy is yet another Python implementation, developed by Armin Rigo et al. and 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 [28].

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 [27].

PyPy actually includes two very distinct modules [25]:

- 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.

2.3.1 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 at 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.

2.3.2 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 (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. It supports six Common Lisp implementations: Allegro CL, Clozure CL, CMU Common Lisp, ECL, LispWorks, and SBCL. 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 [5].

CLPython compiles Python source-code to Common Lisp code, i.e. a sequence of s-expressions. These s-expressions can be interpreted or compiled to `.fasl` files, depending on the Common Lisp implementation used. Python objects are represented by equivalent Common Lisp values, whenever possible, or CLOS instances otherwise. Unfortunately, CLPython does not provide support for C module extensions, since it does not implement the Python/C API [6].

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 [21].

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

2.6 Comparison

Table 2.1 displays a rough comparison between the implementations discussed above.

	Language(s) written	**Platform(s) targeted**	**Speedup (vs. CPython)**	**Std. library support**
CPython (1994-)	C	CPython's VM	$1\times$	Full
Jython (2000-)	Java	JVM	$\sim 1\times$	Most
IronPython (2006-)	C#	CLI	$\sim 1.8\times$	Most
PyPy (2007-)	RPython	C, JVM, CLI	$\sim 6\times$	Most
CLPython (2006-2013)	Common Lisp	Common Lisp	$\sim 0.5\times$	Most
PLT Spy (2003-2005)	PLT Scheme, C	PLT Scheme	$\sim 0.001\times$	Full

Table 2.1: 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 modules extensions, through its use of the Python/C API. However, the performance cost that results from the repeated conversion of data from Scheme's internal representation to CPython's 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 chapters.

Chapter 3

Runtime Environment

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, therefore, 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 chapter will describe the implementation of the runtime environment for PyonR. The compilation process will be described in chapter 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. [39, ch. 3]. 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 chapter 2, we tried two alternative implementations: the first one relies on using a foreign function interface to map Python's operations into foreign calls to the Python/C API [41]; the second one consists of reimplementing Python's semantics and built-in data-types in Racket. This section describes both these attempts.

3.2.1 ...using Racket's Foreign Function Interface

For our first approach, we started by following a similar strategy to PLT Spy, by mapping Python's data types and primitive functions to those provided by 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 [11], and therefore their runtime is implemented in C. This entails converting Scheme values into Python objects and back into Scheme values for each runtime call. Besides the performance issue mentioned in section 2.5, this method lacks portability and is somewhat cumbersome for development, since it requires compiling the runtime module with a platform specific C compiler.

Instead, we used the Racket Foreign Function Interface (FFI) [2] to directly interact with the foreign data types returned by the Python/C API, therefore our runtime is implemented in Racket. The purpose of this FFI is to link Racket with foreign libraries, allowing foreign functions to be called directly from Racket. It automatically converts some C types to their Racket equivalents (e.g. `int` to Racket integers, `char*` to Racket strings) and it supports pointer arithmetic and dereferencing.

We use the FFI to define a Racket interface for the functions provided by the Python/C API, which are then used by our runtime environment. This means that we do not need to define any structures for representing Python objects. The values passed around correspond to pointers to Python objects in CPython's virtual machine. As with PLT Spy, this approach only requires implementing the Python language constructs, because the standard library and other libraries installed on CPython's implementation are readily accessible.

As an example, consider the implementation of the plus operator, as `py-add`:

```
(define (py-add x y)
  (PyObject_CallObject (PyObject_GetAttrString x "__add__")
                       (make-py-tuple y)))

(define (make-py-tuple . elems)
  (let ([py-tuple (PyTuple_New (length elems))])
    (for ([i (length elems)]
          [elem elems])
      (PyTuple_SetItem py-tuple i elem))
    py-tuple))
```

The capitalized function names correspond to Python/C API functions, i.e. foreign functions. First we fetch the `__add__` method from the first argument with `PyObject_GetAttrString`, we pack the second argument into a Python tuple with `make-py-tuple` and we call the method with `PyObject_CallObject`. The `make-py-tuple` function uses `PyTuple_New` to allocate a new tuple with capacity for one object and sets it with `PyTuple_SetItem`. Therefore, we have a total of 4 foreign function calls for a simple addition, which is too expensive.

Indeed, early benchmarks showed that the repetitive use of these foreign functions introduces a significant overhead on our primitive operators, resulting in a very slow implementation [29][30].

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

Another issue with this approach is that it leads to a poor interoperability with Racket, since Python objects have to be explicitly converted to their Racket representations, and vice-versa, when mixing Python and Racket code.

3.2.2 ...using a Racket data model

Due to the issues mentioned above, we experimented with a second approach, inspired by the implementations of Jython, IronPython, and CLPython. This one is a pure Racket implementation of Python's data model. Comparing it to the FFI approach, this one entails implementing all of Python's standard library in Racket, but, on the other hand, it is a much faster implementation and provides reliable memory management of Python's objects, since it does not need to coordinate with another virtual machine.

As mentioned earlier, CPython stores each object in a `PyObject` structure which contains a reference to its type-object. While the same strategy would work in Racket, there is room for improvement. In Racket, one can recognize a value's type through its predicate (`number?`, `string?`, etc.). In Python, a built-in object's type is not allowed to change, so we can directly map basic Racket types to Python's basic types. To name some:

- Python's numerical tower (`int`, `long`, `float`, `complex`) is mapped to Racket numbers;

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

Similarly to CPython's architecture, built-in types without a suitable equivalent in Racket are mapped to subtypes of the `python-object` structure, whose only field is a reference to their type-object. For instance, Python's lists are mutable and also have $\mathcal{O}(1)$ access time. Since the concept of object identity is particularly important in Python, we map Python lists to the `list_obj` structure, which contains a vector. This way, operations which alter a list's size can allocate a new vector, mutating the structure and therefore they do not affect the object's identity.

As mentioned, most Python operations require computing an object's type in order to lookup a method in its hash-table. Since the objects which are directly mapped to Racket data-types do not store a reference to their type-objects, we compute them through a pattern matching function which returns the most appropriate type-object, according to the predicates satisfied by the value. By doing so, we avoid the overhead from constantly wrapping and unwrapping frequently used values from the structures that hold them. Interoperability with Racket data types is also greatly simplified, eliminating the need to wrap/unwrap values when using them as arguments or return values from functions imported from Racket.

3.2.3 Comparison

Putting these two distinct approaches into perspective, the first one allows us to access every library supported by CPython, but, on the other hand, it suffers from two problems: (1) simple operations need to perform a significant number of foreign calls, which leads to an unacceptably slow performance and (2) Python values have to be explicitly converted to their Racket representation when mixing Python and Racket code, resulting in a clumsy interoperability.

By reimplementing Python's semantics and built-in data-types in Racket, we ended up with a much faster implementation, since we can now take advantage of Racket's performance gains. Also, since most Python data-types map directly to the corresponding ones in Racket, interoperability between both languages feels much more natural. On the other hand, this approach requires a greater implementation effort. Additionally, it does not provide us with access to Python libraries based on C module extensions (such as NumPy) by default.

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 Implementing Python's data-types

This section will describe other relevant aspects of this runtime environment's implementation, mainly how we mapped Python's types and their semantics to a Racket data model.

3.3.1 Type-Objects

Python's type-objects encapsulate a specific type's functionality. Each Python object has one and only one type-object. Python programmers can also define their own custom type-objects through class definitions.

A type-object is implemented as a structure (subtype of the `python-object` structure) which holds its name, the name of the module where it was defined, a vector containing the references to its parent type-objects, a documentation string, a hash-table mapping its attribute and method names to their respective objects, and a vector representing a linearization (ordered sequence) of its super types.

Python's type-objects support multiple inheritance and the ordering of its super types is done using the C3 superclass linearization algorithm [1], which they refer to as MRO (Method Resolution Order).

We compute this linearization once, during the type-object's initialization, from its parent types and store it in the type-object. Python uses duck typing, therefore this linearization is used to specify the order in which an object's super types are looked up when dispatching an attribute or method.

As mentioned earlier, to obtain an object's type-object we rely on a simple pattern matching function. An excerpt of its implementation is shown below:

```
1 (define (type x)
2   (cond
3     [(number? x) (number-type x)]
4     [(string? x) py-string]
5     [(python-object? x) (python-object-type x)]
6     [(vector? x) py-tuple]
7     ...
```

It can be seen that strings are trivially recognized as the `str` type (defined as the `py-string` variable). The same can be seen for vectors, which are recognized as the `tuple` type. For numbers, a more specific function is dispatched, which returns the types `int`, `long`, `float`, or `complex`. Objects represented by the `python-object` structure hold a reference to their type-object, which is accessed by the `python-object-type` selector.

The functionality for a given type is stored on the type-object's hash-table. This hash-table maps method names to the functions which implement them. Instead of storing the method names as strings, we chose to use Racket symbols, which act as interned strings. This means that two symbols which the same content will always have the same identity. This way, we can use identity hash-tables (Racket's

hasheq) instead of equality hash-tables (Racket's hash), thus comparing keys with eq? (identity comparison) instead of the more expensive equal? (equality comparison).

These symbols are still presented to the user as strings when he inspects a type-object's dictionary or changes its content dynamically, which entails converting them with symbol->string and string->symbol. Nonetheless, since reading entries from these hash-tables is far more frequent than changing them, the time spent on symbol to string conversion is negligible compared to the performance gained from hashing symbols instead of strings.

Summing up, to dispatch an object's method, the object's type-object t is first computed with the type function, then that method's name is looked up in the hash-table of each type-object u in t's MRO linearization, until it is found. If the method's name is not present in u's dictionary for any u, a TypeError exception is raised.

3.3.2 Functions and Callable Objects

Python's functions (defined by the def or lambda keywords) are quite similar to Racket's functions in the sense that they are both first-class citizens and are defined in the same namespace as other variables (Racket is a *Lisp-1*, in Lisp terminology). However, Python's functions must also store the ordered names of their parameters, since their arguments can both be called by position or by keyword.

Racket structures can be defined to implement callable semantics, with the prop:procedure property [12, ch. 4.17]. We take advantage of this to store a Python function object as a function_obj structures (a python-object substructure) which holds a procedure and the list of its parameter names. We use the prop:procedure property to specify that a call to a function_obj should call the stored procedure instead.

In addition to function objects, any Python object can be made callable by defining a __call__ method in its type-object. To cope with this, our python-object structure also implements a prop:procedure property, which dispatches this __call__ method when attempting to call an instance of this structure. If this method is not defined, this raises a TypeError signalling that the object is not callable.

3.3.3 Exceptions

Our exception objects share two representations: the standard exn structure used by Racket exceptions [12, ch. 10.2] and a python-object substructure which, like Racket's, holds a slot for a message string and a slot for continuation marks (Racket's implementation of a stack-trace).

The rationale for defining our own structure for exceptions is simple: we wanted to replicate Python's class hierarchy for exceptions, which could not be mapped to Racket's structure hierarchy because they are too different. In this case, each exception's type is stored in python-object's slot for the object's type.

We chose to also recognize Racket exceptions as Python exceptions so that we could reuse Racket's functions without implementing additional safeguards. For instance, Racket's number division function raises the exception exn:fail:contract:divide-by-zero when the quotient is zero. Python has a sim-

ilar behaviour for its division operator used on numbers, but raises the exception `ZeroDivisionError`.

To implement this, instead of testing whether the quotient is zero ourselves and raising our own `ZeroDivisionError` exception, we chose to simply call Racket's number division function. This way, reimplementing Python's standard library is much easier and we also improve the general performance of these functions, because Racket will enforce these safeguards, whether we also do it ourselves or not.

In order to have these Racket exceptions recognized as the corresponding Python exceptions by the exception handling constructs, our `type` function dispatches the `racket-exception-type` function when it finds an instance of the `exn` structure. This function simply maps Racket exceptions to the type-objects we use for Python exceptions. For this case, the `exn:fail:contract:divide-by-zero` exception raised by Racket's number division function is recognized as the `ZeroDivisionError` type-object. If no rule applies to a specific exception, the default case will return an umbrella type for Racket exceptions: `RacketException`.

The only drawback to this is that the message strings produced by these exceptions are not exactly the same as the ones used by Python's reference implementation. Still, they should be sufficiently explicit, since they are used for similar purposes.

A similar approach is also followed by Clojure, a dialect of Lisp for the JVM. Instead of implementing their own safeguards similar to, for instance, Common Lisp's condition system, they use Java's exceptions.

3.3.4 Miscellaneous Racket Values

The default case for our `type` function is the `racket_value` type-object. This is nothing more than an umbrella type-object for Racket values, which only implements the `__repr__` method, responsible for specifying how an object should be printed. This is implemented as their external Racket representation.

This default case is only reached by Racket values which are not supposed to have a Python mapping and therefore are inaccessible from Python itself, unless they are explicitly imported from Racket libraries, as made possible by the import mechanisms which will be described in section 5.2.

3.4 Optimizations

This final section will describe some performance optimizations we have implemented, mostly to take advantage of Racket's data model. The performance gains from each of these optimizations will be later showcased on chapter 7.

3.4.1 Early Dispatch

Despite the ability to add new behaviour for operators in user-defined classes, a typical Python program will mostly use Python's arithmetic operators for numbers and occasionally strings. Since most of these operations are supposed to be very quick (in lower level languages they are compiled directly to CPU instructions), the overhead imposed by Python's heavy dispatching mechanism becomes too much.

Therefore, each operator implements an early dispatch mechanism for the most typical argument types, which skips the heavier dispatching mechanism described above. For instance, instead of implementing the plus operator as:

```
(define (py-add x y)
  (py-method-call x "__add__" y))
```

Where the `py-method-call` macro implements Python's method dispatching semantics. We now implement it as:

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

This makes the plus operator run nearly as fast as a standard Racket number addition or string concatenation operation, for numbers and strings, respectively, while still respecting Python's semantics for operator customization. Besides the plus operator, this optimization encompasses all unary and binary operators and comparisons, for the relevant types.

3.4.2 Sequence Iteration

Python's `for` statements, list comprehensions, and some built-in functions like `min` and `max` all rely on getting an iterator object (made available by the `__iter__` method) from the sequence about to be iterated. This iterator object must support a `next` method which returns the next element in the sequence or raises a `StopIteration` exception to signal its exhaustion. This way, a user-defined class may specify an `__iter__` method which returns an object with a `next` method, so that objects from this user-defined class may be iterated with a `for` statement or similar construct.

The issue with implementing this in Racket is that installing an exception handler is an expensive operation. Since most uses of the `for` statement in a typical Python program will be for built-in objects, we can take advantage of their internal representation to come up with more efficient ways to determine whether there are still elements to iterate and which is the next one.

Racket has a built-in concept of iterables, called sequences [12, ch. 4.14.1]. Many built-in Racket data-types are sequences by default, including lists, vectors, and strings. Additionally, user-defined structures can be recognized as sequences by implementing the `prop:sequence` property. One can use the Racket function `sequence-generate` on a sequence, which returns two procedures: the first one returns true if there are still available values and the second one returns the next element from the sequence.

Therefore we have implemented a `->sequence` function which returns a sequence, given a Python iterable.

```
(define (->sequence obj)
    (match obj
      [(? list_obj?) obj]                          ;; Python lists
      [(? vector?) obj]                            ;; Python tuples
      [(? string?) (sequence-map string obj)]      ;; Python strings
      [(? dict?) (in-dict-keys obj)]               ;; Python dicts
      [(? python-set?) obj]                        ;; Python sets
      [_ (py-obj->sequence obj)]))
```

For Python lists, our `list_obj` structure already implements the `prop:sequence` property, therefore we can simply return them back. Python tuples and sets are also returned back since their Racket representations are sequences by default (tuples are Racket vectors and sets are Racket custom sets). For strings, we must build a new string containing each iterated character, since there is no data-type for characters in Python; instead characters are strings with a length of one. Iterating through a dict, yields each one of its keys.

If the iterated object is not one of these types, the function `py-obj->sequence` returns a procedural sequence which implements the standard Python mechanism described above. This way we can still support customizing user-defined classes to be iterable, but with a slightly slower implementation nonetheless. Still, we believe this is a tradeoff worth paying for due to the huge performance gains we get by simplifying the iteration over built-in types.

3.4.3 Attribute Getting and Setting

In Python, to get an attribute `x` from an object `foo`, one would type `foo.x`. This roughly expands to `type(foo).__getattribute__(foo, 'x')`. Like other operators, its behaviour is customizable by overriding the `__getattribute__` method, but no built-in types do so, therefore the default method (defined by the `object` type is usually the one dispatched for this dot notation.

This method implements a rather sophisticated algorithm. For the example above, it follows these steps:

1. The attribute name `"x"` is looked up in `foo`'s type-object's dictionary and its supertypes. If an object is found and it is a data descriptor (i.e. it implements a pair of `__get__` and `__set__` methods), Python dispatches a call to that data descriptor's `__get__` method;

2. Else, if `foo` has an instance dictionary (present in module objects and objects instantiated by a user-defined class) and it contains an entry for `"x"`, Python dispatches that object;

3. If an attribute was found on step 1, but it is a non-data descriptor (i.e. it only implements a `__get__` method), Python also dispatches a call to that descriptor's `__get__` method;

4. If an attribute was found on step 1, but it is not a descriptor, Python dispatches that attribute;

5. Else, an `AttributeError` exception is raised.

Assigning a value to an object's attribute is similar in every way. For instance, to set the value `4` to `foo`'s `x` attribute, one would type `foo.x = 4`. This expands to `type(foo).__setattr__(foo, 'x', 4)`. The default method follows these steps:

1. The attribute name `"x"` is looked up in `foo`'s type-object's dictionary and its supertypes. If an object is found and it is a data descriptor, Python dispatches a call to that data descriptor's `__set__` method;

2. Else, if `foo` has an instance dictionary (present in module objects and objects instantiated by a user-defined class), an entry for `"x"` is set on the dictionary;

3. If an attribute was found on step 1, but it is a non-data descriptor, an `AttributeError` exception is raised (the attribute is read-only);

4. Else, an `AttributeError` exception is raised (the attribute does not exist).

In order to optimize the performance for these two methods while still retaining their correct semantics, we added two extra fields to the type-object structure: `getter` and `setter`. The `getter` field is initialized with the object's `__get__` method if it exists, or false otherwise. Similarly, the `setter` field is initialized with the `__set__` methods if both the `__get__` and `__set__` methods exist, or false otherwise.

This way, by obtaining these fields, we can quickly determine if an object is a data or non-data descriptor by checking their logical values and we can immediately dispatch the corresponding method.

One other optimization was to develop two separate implementations for each method: one for objects with an instance dictionary and another, which skips step 2, for those without it. This allowed us to greatly simplify the logic on each method, leading to less conditions to be tested and, therefore, faster implementations.

Chapter 4

Compilation

This chapter 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).

Fig. 4.1 summarizes the pipeline for compiling and interpreting Python code on PyonR:

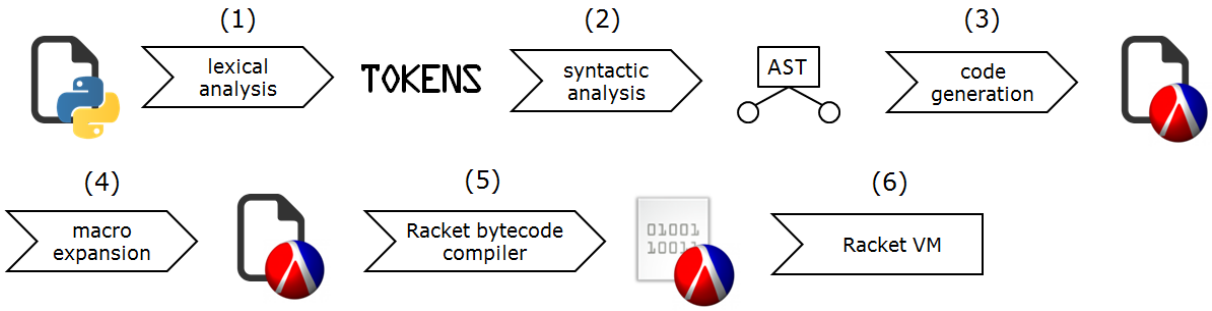


Figure 4.1: PyonR's pipeline for interpreting Python source-code

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
```

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`. Note that this does not compromise portability across different Python implementations because the hash character starts a line comment in Python.

The following lines are parsed according to the reader module specified by `#lang python`. For this example:

- 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 [18] for producing a scanner and a Yacc specification [15] for producing an LALR(1) parser, both of which are implemented fully in Racket, using the `parser-tools` library [24]). The grammar originally used by PLT Spy and ported to `parser-tools` is described at [33].

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).

4.1.1 Alternative Approaches

Apart from implementing our own scanner and parser, we considered another alternative for scanning and parsing Python code: using Python's `ast` library [40, ch. 31.2] in a separate CPython process to parse the Python code and produce a Python AST object, then somehow convert it to a Racket representation. This could be done, for instance, by compiling the AST to a textual format such as XML, reading it from Racket through a pipe, and rebuilding it back into a tree-like Racket structure;

In the end, we opted against it for a number of reasons:

- Even though the parser in Python's `ast` library is complete and bug-free by definition, the process of translating the Python AST objects into another representation and back into a Racket AST would be tedious and not guaranteed to be bug-free;
- PLT Spy's scanner and parser were fully written in PLT Scheme, which is easy to port to Racket.
- Since Python's syntax has not changed much from version 2.3 to 2.7, changes would be easy to handle.
- We later came up with the need to extend Python's syntax in order to support different ways to import modules (as explained in chapter 5), which would be very hard to integrate with Python's `ast` library.

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 purely 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 [10, ch. 16.2.1] are Racket's built-in data type for representing code. They contain the quoted form of the code (an s-expression), source location information (line number, column number and span) and lexical-binding information. By keeping the original source location information on every syntax-object generated by the compiler, DrRacket can map each compiled s-expression to its corresponding Python code. This is crucial for taking advantage of most of DrRacket's features, as will be explained in section 6.1.

The rest of this section will describe the decisions regarding code generation for the most relevant nodes.

4.2.1 Literals

Python has literal notations for numbers (`int`, `long`, `float`, and `complex`) and for strings. As mentioned in chapter 3, PyonR maps both Python's numbers and strings to Racket's numbers and strings, respec-

tively. Therefore, generating their Racket code is as simple as returning the values stored inside their nodes.

4.2.2 Identifiers

Python identifiers (variable names) are also compiled to Racket identifiers, i.e. unquoted symbols. However, their compiled names are prefixed by a colon. We do this for two reasons:

- Racket supports hygienic macros, but the syntax-objects generated from traversing the AST nodes purposely break hygiene. By mangling Python identifier names, we avoid shadowing our Racket primitives with user-defined names. For example, a user might set a variable `cond` in Python, which will then be compiled to `:cond` and therefore will not shadow Racket's built-in `cond` that may also appear in the compiled code.

- Most importantly however is that since `#lang python` provides Racket functions only meant to be used in compiled code (e.g. `py-add`), by mangling every identifier name in Python with a prefixed colon, we are making sure that these Racket names are unreachable from Python, therefore ensuring that implementation details do not leak into Python user code unless explicitly imported.

4.2.3 Assignments

Python uses the same assignment syntax for defining a new variable and for setting it. This is quite different from Racket's assignment model. Since Racket performs constant inlining by default when it can prove that a variable is not re-assigned (i.e. when it is a constant), a variable must first be defined with a `define` form and later changed with a `set!` form.

Additionally, this syntax is very flexible, allowing not only the setting of variables (e.g. `a = 3`), but also, for instance, setting object attributes (e.g. `b.x = 4`), mutating collections (e.g. `L[10] = 5`), or even combine multiple assignments, drawing values from a collection (e.g. `(a, (b.x, L[10])) = [3, [4, 5]]`).

We trivially compile all assignments to the macro form `(py-assign!` $target$ $value$`)`. Thus, most of the work is done at macro expansion time, by recursively expanding the `py-assign!` macro according to the pattern that follows.

If $target$ is an explicit list or tuple, the form is expanded to a sequence of other `py-assign!` forms, one for each element of the list/tuple, where the assigned value is the result of iterating over $value$.

If $target$ is a attribute referencing, item getting or slice getting operation, the whole form is expanded to an attribute setting, item setting or slice setting operation, respectively. Finally, if $target$ is an identifier, the form is expanded to a `define` or `set!` form, depending whether the identifier already has a binding (Racket macros handle syntax-objects, thus we are able to compute their lexical binding information at macro-expansion time).

4.2.4 Conditional Statements

In Racket, only `#f` (the Boolean value false) is treated as false when used as the condition of an `if` or other conditional forms. In Python, the Boolean value false, zero, and any object whose length returns zero (empty string, empty list, empty dictionary, etc.) are all treated as false.

Thus, we compile the condition on `if` and `while` statements as `(py-truth` *condition*`)`. The function `py-truth` takes a Python object as argument and returns a Racket boolean value, `#t` or `#f`, according to Python's semantics.

4.2.5 Function Definitions

In Python, named functions are defined with the `def` keyword. Python also supports defining anonymous functions as expressions with the `lambda` keyword, just like Racket.

Function and class definitions are the only statements in Python which introduce a new lexical scope, i.e. when assigning variables inside a function definition's body, these variables are assigned in a local namespace that shadows any variables with the same name assigned in an outer scope. Optionally, Python programmers may specify that a variable name inside a function definition refers to the variable defined in the global namespace, using the `global` statement. This way, further assignments to that variable will be global assignments.

We implemented this by collecting every variable name defined inside the function's body, except for the function's parameters and variables specified with `global`, and defining them locally in a `let` form. Racket's `let` form introduces a new lexical scope, therefore assignments for these variables inside the function definition's body will compile to `set!` forms, which act on the bindings defined by `let`.

One other concern with function definitions are keyword arguments. In Racket, a function's parameter is defined as either positional or by keyword. In Python, a function can be called both with positional or keyword arguments, or a mix of both, as every parameter is simultaneously positional and by keyword. Therefore, when compiling a Python function definition, we define the parameters as positional, but we also store the list of parameter names in a callable structure, using our own `(define-py-function ...)` macro.

Through this, we are able to support the use of keyword arguments for user-defined functions: if the function is called with keyword arguments, the call is handled by the `py-call/keywords` macro, which rearranges the arguments' order at runtime.

If the function is called without keyword arguments (the most commonly used variant), it can be handled directly as a Racket function call, without additional overheads. This way, we can use the same syntax for calling both Python user-defined functions and imported Racket functions (as will be seen in section 5.2).

The last things worth mentioning are the `return` and `yield` statements. The `return` statement immediately returns to the point where the function was called, returning a given value. This feature is present in most modern programming languages (including Python) but not in Racket. Since every form in Racket is an expression, i.e. it produces a value, the return value of a function call is the value of its

last form. With this in mind, we can separate Python's `return` statement use into two categories:

- Tail returns – when the `return` statement is not followed by other statements on its control flow;
- Early returns – when the `return` statement is followed by additional statements on its control flow, which will not be executed because the `return` statement causes execution to leave the function's body.

We implemented early returns as escape continuations, with Racket's `let/ec` form (syntactic sugar for a `let` and a `call-with-escape-continuation`) [12, ch. 10.4] wrapping the body of the function definition. With this approach, compiling a return statement is as straightforward as calling the escape continuation.

Since capturing continuations is a rather expensive operation, we only use `let/ec` when there is at least one early return. If all `return` statements are tail returns, they can be trivially compiled to the expression they are returning, since they will be the last expressions on their control flow.

As for the `yield` statement, calling a function whose definition contains a `yield` statement does not result in that function's execution; instead, the function call returns a generator object. This generator encapsulates the function's execution context. Generators have a `next` method which results in the execution of the function's body until the `yield` statement, returning its value, similarly to a `return` statement. However, when the `next` method is called again, its execution resumes from the context where it previously left.

This behaviour can also be implemented through the use of continuations to capture a function's execution context, but, fortunately for us, Racket already supports a similar abstraction over continuations with its own generators and its `yield` form [12, ch. 4.14.3]. Thus, to implement Python's `yield` statement, we simply have to map it to a `yield` form and compile the function definition's body within a `generator` form.

4.2.6 Class Definitions

A class definition, using the `class` statement is syntactic sugar for assigning a type-object to a variable, similarly to how a `def` statement is syntactic sugar for assigning a function object to a variable.

Like function definitions, a class definition introduces a new lexical binding. Again, we implemented this by collecting every assigned variable name in the class definition's body and declaring them locally in a `let` form. These include method definitions, nested class definitions and other local assignments.

The class statement's body is then compiled inside the `let` form, so that these assignments will act on the local `let` bindings. Finally, a new type-object is instantiated by using the name and super-types declared in the head of the `class` statement and instantiating a new dictionary made up of the collected variable names and their bindings. This type-object is assigned to the class name with `py-assign!`.

4.2.7 Loops

Python supports two looping constructs: `for` and `while`. Both of them allow the use of the `break` and `continue` keywords. The former immediately leaves the cycle, while the latter simply ends the current

iteration to start the next one.

Python's `while` cycle is similar to that of languages such as C or Java. It repeatedly evaluates its body as long as its condition holds true. It is compiled to a named `let` form.

A named `let` form [10, ch. 4.6.4] defines a local function and implicitly calls it with some initial arguments. When it is used as a tail call, it acts as a `goto` with arguments [34]. The `while` cycle is implemented by a named `let` form which evaluates the condition and if true, evaluates the body and calls itself. Note that calling this named `let` has the same semantics as a `continue` statement, i.e. starting the next iteration. In fact, `continue` statements are compiled to a call to our named `let` form (which predictably is named `continue`).

Python's `for` cycle is used to iterate over collections or other objects that support iteration, such as generators. It is similarly compiled to a named `let`. This one updates the control variable by getting the next value from the iterator, evaluates the statement's body and recursively calls itself, repeating the cycle with the next iteration. The `continue` statement has the same semantics and, as such, can be implemented the same way.

Finally, a `break` statement is implemented just like the `return` statement described above. We use a `let/ec` form wrapping the named `let` form if the cycle contains a `break` statement. The `break` statement itself is compiled to a call to the escape continuation.

4.2.8 Raising and Handling Exceptions

Both Python and Racket support exceptions in a similar way. In Python, exceptions are raised with the `raise` statement and caught with the `try...except` statement (with optional `else` and `finally` clauses).

We have implemented Python's exception handling over Racket's. The `try...except` statement is mapped to a `with-handlers` form [12, ch. 10.2]. This expects an arbitrary number of pairs of predicate and procedure. Each predicate is responsible for recognizing if a specific exception should be caught and the procedure specifies how that exception is handled. Each Python `except` clause contains the exception type(s) it should handle and a body specifying how to handle them.

Each `except` clause is compiled to a `with-handlers` pair: a predicate that recognizes if a given exception is a subtype of the declared exception types (by using the linearization computed by the MRO) and a `lambda` form containing the body of the `except` clause. The body of the `try` clause is compiled to the body of the `with-handlers` form.

A Python `raise` statement is mapped almost directly to a Racket `raise` form [12, ch. 10.2]. Python's `raise` statement can be used with an exception object or type-object. In the latter case, a new exception is instantiated and raised, capturing the current stack-trace with Racket's `current-continuation-marks` function.

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. As a simple example, consider a naive implementation of the Fibonacci sequence:

```
1  def fib(n):
2    if n == 0: return 0
3    elif n == 1: return 1
4    else: return fib(n-1) + fib(n-2)
5
6  print fib(10)
```

With our original approach, this is compiled to:

```
1  (define-py-function :fib (n)
2    (lambda (:n)
3      (cond
4       ((py-truth (py-eq :n 0)) 0)
5       ((py-truth (py-eq :n 1)) 1)
6       (else (py-add (:fib (py-sub :n 1)) (:fib (py-sub :n 2)))))))
7
8  (py-print (:fib 10))
```

In order to run this code, we are dependent on the definitions of the macro `define-py-function` and the functions `py-truth`, `py-eq`, `py-add`, `py-sub`, and `py-print`. Likewise, this piece of code is not particularly readable nor maintainable.

This new translator compiles it to the following:

```
1  (define (fib n)
2    (cond
3     ((= n 0) 0)
4     ((= n 1) 1)
5     (else (+ (fib (- n 1)) (fib (- n 2))))))
6
7  (displayln (fib 10))
```

Similarly to how we override the `to-racket` method on every AST node, we now have a similar `to-readable-racket` method. Among the differences:

- Function definitions are compiled to standard `define` forms. We lose the ability to call these defined functions by keyword argument, but, for all purposes, positional and keyword arguments do not coexist in Racket;
- Conditional statements are not safeguarded by `py-truth`. This is based on the expectation that the expressions used for conditions will mostly return Boolean values.
- Binary expressions compile to their Racket numerical counterparts. This is based on the expectation that they will be mostly used with numbers. Since these operators are typically overloaded for each type, future work could involve inferring the types of the operands at compile-time in order to figure out the most appropriate compilation result. For instance, Python's plus operator on strings performs string concatenation. On the Python expression `x + "abc"`, the string literal `"abc"` hints that `x` is also a string and the plus operator refers to a string concatenation, therefore this should compile to `(string-append x "abc")`.
- The `print` statement is compiled to a `displayln` form.
- Identifiers are no longer prefixed by colons. Since the compilation results are to be inspected by programmers, it becomes easy to detect the unintended shadowing of identifiers. Additionally, this enhances readability.

Chapter 5

Interoperability

PyonR would not be complete without supporting Python's `import` statements, used to import functionality from other Python files (**Fig. 5.1**, blue arrow). Additionally, it would have a very limited practical use if one could not also import Racket modules from Python, particularly Rosetta, and Python modules from Racket (**Fig. 5.1**, green arrows). Finally, we also implemented an approach for importing functionality from CPython's virtual machine (**Fig. 5.1**, red arrows), which is particularly useful for reaching Python's standard library or other C based libraries only available to CPython.

This section will describe the problems faced, design decisions taken, and results obtained for these three challenges.

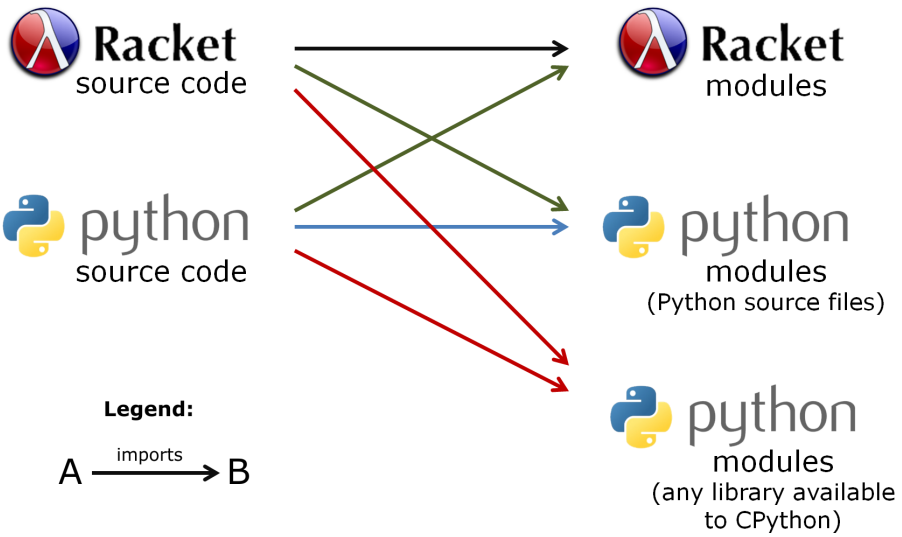


Figure 5.1: Overview of the possibilities for interoperability offered by PyonR

5.1 Interoperability 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 in <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.

Names starting with an underscore are considered private and will not be imported using the first and third syntaxes, but they can be explicitly imported using the second one.

5.1.1 Locating modules

The `sys` module exports a `path` variable which contains a list of path strings where the interpreter will look for the specified module. This list is initialized with some default paths and it can be changed at runtime, thus allowing files in other directories to be imported.

Consider we wanted to import a file named `foo.py`, located at `/path/to/module/`. The typical way to import this file is:

```
1 import sys
2 sys.path.append("/path/to/module/")
3 import foo
```

5.1.2 Implementing the `import` syntaxes

In Racket's case, bindings from other modules are imported using the `require` form. Furthermore, one can only require bindings which have been explicitly provided by that module, using the `provide` form. Each of these forms takes a specification for which bindings to be `require`'d or `provide`'d, allowing the user to filter and/or rename them in a very expressive way [12, ch. 3.2].

The `require` form introduces its bindings during macro-expansion time (i.e. at compile time). This way, Racket's macro expander can always track the origin of an identifier before running the code and will signal a syntax error if an identifier is unbound. Additionally, we can also get a module's binding at runtime using the `dynamic-require` function [12, ch. 14.4.3], which returns the imported binding.

Since the location of a module must be determined through the `path` variable at runtime, the importing of said module must happen at runtime as well. Therefore, all three syntaxes were implemented through dynamic requiring.

For the first one (`import...as`), we make use of `module->exports` [12, ch. 14.4.3] 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 (module objects are implemented as a substructure of `python-object`). For the second one (`from...import...as`), 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.

Since the `path` variable cannot be reached at macro-expansion time to locate the specified module, we chose to use default paths that are used to initialize the `path` variable. While this stands as a limitation to this import statement, there is still a practical way to overcome it. These default paths are stored in a text file and loaded during the macro-expansion phase, therefore the user can still populate this file with useful paths so that the `from...import*` statement will find the intended modules.

As for declaring and visiting the module, we use `namespace-require/expansion-time` [12, ch. 14.1]. 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.

As an alternative, we could do the import fully at runtime (including locating the module) by adding the imported bindings to the current namespace (using `namespace-set-variable-value!` [12, ch. 14.1]) instead of defining new bindings for them. To suppress the syntax errors from unbound identifiers, we could similarly override the `#%top` macro to directly lookup the identifier's name in the current namespace (using `namespace-variable-value` [12, ch. 14.1]).

Doing so, however, would eliminate the signalling of syntax errors for unbound identifiers altogether, since the point of failure would be at runtime, when `namespace-variable-value` fails. We chose not to accept this tradeoff mainly because (1) the `from...import*` statement is as powerful as the others, thus being easily replaced by the user if needed, and (2) the ability to detect unbound identifiers at compile-time is a great tool for avoiding bugs, whether they result from misspelled names or from a wrong use of scoping. When using DrRacket, this feature is additionally useful to improve code readability since it can track and display the origin of each bound identifier. This feature will be explained in detail in section 6.1.

Finally, it is worth mentioning that since every definition in a Python file can be imported, as well as other imported bindings, our `#lang python` modules `provide` all definitions and all bindings from required modules.

5.2 Interoperability 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 (line 2), arbitrary files (line 3), collections from PLaneT: Racket's centralized package distribution system (line 4), etc.

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

5.2.1 Name mangling

One thing worth taking into account when importing Racket identifiers to Python is that the set of allowed identifier names in Python is a subset of those in Racket. In Python, an identifier must start with a lowercase letter, uppercase letter or underscore, followed by zero or more letters, underscores or digits, i.e. they must match the regular expression `[a-zA-Z_][a-zA-Z_0-9]*`.

In Racket, any character may appear in an identifier, except for whitespace and the special characters `( ) [ ] { } " , ' ` ; | \`. For instance, the expression `3+2` is a valid identifier, while in Python this would be interpreted as the addition of the integers `3` and `2`.

Additionally, one can include whitespace or special characters in an identifier by quoting them with vertical slashes. This can also be used to quote numbers so they are read as identifiers. Consider the following interaction as an example:

```
> (define |one, two, three| (list 1 2 3))
> (define |6| 6)
> (cons |6| |one, two, three|)
'(6 1 2 3)
```

As for naming conventions, Python names are usually lowercase (except for class names, which are capitalized) and the words are separated by underscores. In Racket, identifier names are also usually lowercase, but words are separated by hyphens.

With this in mind, we attempted to make most Racket identifiers reachable in Python through name mangling. We use the set of rules listed on table 5.1 when importing identifiers:

Going back to the example above, the identifier `3+2` would be imported to Python as `_3_PLUS_2`. We deliberately chose to break the naming convention with the use of capitals in order to avoid possible collisions among converted names (e.g. consider the not so far fetched example of a library exporting an identifier with a name followed by `=` and another followed by `-equal`).

Trigger	Rule
contains "->"	replace with "TO"
contains "<-"	replace with "FROM"
contains "!"	replace with "BANG"
contains "$"	replace with "DOLLAR"
contains "%"	replace with "PERCENT"
contains "&"	replace with "AND"
contains "*"	replace with "STAR"
contains "+"	replace with "PLUS"
contains "_" (underscore)	replace with "UNDERSCORE"
contains "."	replace with "DOT"
contains "/"	replace with "SLASH"
contains ":"	replace with "COLON"
contains "<="	replace with "LE"
contains ">="	replace with "GE"
contains "<"	replace with "LT"
contains ">"	replace with "GT"
contains "="	replace with "EQUAL"
contains "?"	replace with "QUERY"
contains "@"	replace with "AT"
contains "^" (caret)	replace with "CARET"
contains "~" (tilde)	replace with "TILDE"
begins with "-" (hyphen)	replace with "MINUS"
contains "-" (hyphen)	replace with "_" (underscore)
begins with digit	prefix with "_" (underscore)

Table 5.1: Name mangling rules

This set of rules does not fully guarantee a mapping from every possible Racket name to a valid Python name, particularly for names with Unicode characters or those quoted with vertical slashes. However, such names are extremely rare or non-existent in Racket libraries, so this mapping should be enough for the vast majority of cases.

Even so, should a user need to import a Racket binding for which this mapping is not enough, we suggest one of two alternatives:

- Defining a Racket module which requires these bindings and simply provides them with another name, then importing this Racket module instead;
- Importing the module using the first syntax (`import...as`) and accessing the wanted binding using reflection. For instance, consider a Racket module which provides the binding `|one, two, three|`. After importing that module as `foo`, one could access it by typing `foo.__dict__["one, two, three"]`.

5.2.2 Using macros

The first syntax (`import...as`) is implemented like its Python equivalent, with `module->exports` and `dynamic-require`. The main difference is that each binding has each name mangled according to the procedure described above, before being stored in the module object.

The second (`from...import...as`) and third (`from...import*`) syntaxes are implemented with standard `require` forms, since this time there is no need to determine the module's location at runtime.

Imported names are also mangled and prefixed by a colon, by using the `filtered-in` and `prefix-in` specifications [12, ch. 3.2], respectively.

One neat side effect of using `require` instead of `dynamic-require` is that Racket macros can be imported and used to some extent. The program below exemplifies the use of the `trace` macro [12, ch. 18.5.1].

```
1 #lang python
2 from "racket/trace" import trace
3
4 def factorial(n):
5   if n == 0: return 1
6   else: return n * factorial(n-1)
7
8 trace(factorial)
```

In this example, the `trace` macro is being used to redefine the factorial function so that each call and its result is displayed.

The interaction below shows the results of calling the "traced" factorial function in the REPL:

```
> factorial(4)
>(:factorial 4)
> (:factorial 3)
> >(:factorial 2)
> > (:factorial 1)
> > >(:factorial 0)
< < <1
< < 1
< <2
< 6
<24
24
```

5.2.3 Types vs. predicates

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, developers may define predicates which recognize values that adhere to a conceptual type. For instance, the list `'(a b c)` is recognized by the `list?` predicate, but also by the `pair?` predicate. The empty list, `'()`, is recognized by `list?` and `empty?` but not by `pair?`. Likewise, a conceptual type's functionality is not encapsulated in a concrete type, but spread out among global functions.

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*. This was implemented by keeping an association list at runtime, which maps predicates to type-objects. The `type` function recursively tries each key (i.e. each predicate) on the object and returns the mapped value when a predicate matches.

We made this available to the user through a Python module `predicates` which exports the functions:

- `set_predicate(pred, type)`, takes a predicate and the associated type-object and stores that mapping;
- `remove_predicate(pred)`, takes a stored predicate and removes its mapping from the association list.

As a practical example of this feature, consider the three-dimensional coordinates data-type used in Rosetta [19]. It represents a point in space and can be constructed using Cartesian coordinates, with the `xyz` function, or cylindrical coordinates, with the `cyl` function, among others. One can obtain a point's XYZ coordinates with `cx`, `cy` and `cz` or its cylindrical coordinates with `cyl-rho`, `cyl-phi` and `cyl-z`. Additional operations include adding a displacement to a point, by specifying the displacement using Cartesian or cylindrical components, with `+xyz` or `+cyl`, respectively. Finally, this data-type can be recognized by the `position?` predicate.

In Python it would make sense that obtaining a point's coordinates would be done as an attribute referencing operation. We could also generalize the displacement addition operation to add any two positions (where the second position is interpreted as the displacement vector), thus integrating it with Python's plus (+) operator. Finally, we could specify an external representation for sets of coordinates, which differs from the Racket one.

This would be done like this:

```
1  #lang python
2  from '(planet aml/rosetta)' import *
3  import predicates
4
5  class XYZ(object):
6
7      x = property(cx)
8      y = property(cy)
9      z = property(cz)
10
11     rho = property(cyl_rho)
12     phi = property(cyl_phi)
13
14     def __add__(self, other):
15         return PLUS_xyz(self, cx(other), cy(other), cz(other))
16
17     def __repr__(self):
18         return "<" + str(cx(self)) + ", " \
19                    + str(cy(self)) + ", " \
20                    + str(cz(self)) + ">"
21
22 predicates.set_predicate(position_QUERY, XYZ)
```

The `property` function, used on lines 7-12, is a built-in Python function that takes a getter function (and optionally a setter function) to build a descriptor (cf. section 3.4.3). In this case, these are non-data descriptors, i.e. read-only descriptors, which will call the selectors `cx`, `cy`, etc. when reading the attribute `x`, `y`, etc. from an `XYZ object`.

In lines 14-15, we define an `__add__` method based on the imported `+xyz` function (whose name was mangled to `PLUS_xyz`). Similarly, in lines 17-20 we specify a custom representation for this data-type by defining a `__repr__` method.

Finally, in line 22, we map the `position?` predicate to the newly defined `XYZ` class. This way, Racket values recognized by `position?` will be treated as instances of the `XYZ` class.

Here is an interaction example showcasing these new features for the XYZ type:

```
> a = xyz(3,4,8)
> a
<3, 4, 8>
> type(a)
<type 'XYZ'>
> a.x
3
> a.rho
5
```

Let us now add a cylindrical displacement to this position:

```
> from "racket/math" import pi
> b = cyl(6, pi/3, -2)
> b
<3.000000000000001, 5.196152422706632, -2>
> a + b
<6.000000000000001, 9.196152422706632, 6>
```

5.2.4 Dealing with overlapping predicates

One of the issues with predicate dispatch is that 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 have included a third function in the `predicates` module:

- `define_subtype(pred, parent)`, defines the predicate $pred$ as being a subtype of predicate $parent$, i.e. $pred(x) => parent(x)$.

These mappings are stored on a hash-table, meaning that each predicate can only have one parent. When `define_subtype` or `set_predicate` is used, the $(pred, type)$ association list is stably sorted so that if $pred1$ is a subtype of $pred2$, then $pred1$ will show up before $pred2$. In addition, subtypes are transitive, i.e. if $pred1$ is defined as a subtype of $pred2$ and $pred2$ is defined as a subtype of $pred3$, then $pred1$ is a subtype of $pred3$. This means that, in the event of an overlapping set of predicates, our implementation will dispatch the most specific one.

As an example, consider the following sequence of Python interactions (the `List` and `EmptyList` class definitions have been omitted for brevity sake):

```
> from "racket" import empty_QUERY as is_empty, \
                       list_QUERY  as is_list,  \
                       list        as rkt_list
> import predicates as preds
> preds.set_predicate(is_empty, EmptyList)
> preds.set_predicate(is_list, List)
> type(rkt_list())
<type 'List'>
```

The `is_empty` and `is_list` predicates overlap for the empty list. Since the most recently entered one was `is_list`, the `List` type is dispatched. Let us define `is_empty` as a subtype of `is_list`:

```
> preds.define_subtype(is_empty, is_list)
> type(rkt_list())
<type 'EmptyList'>
```

As expected, the `EmptyList` type is now dispatched instead.

As a final comment, it is worth mentioning that the user is responsible for ensuring that the subtype hierarchy is consistent, i.e. there are no cycles. Given the transitivity rule, if there is a cycle among subtypes, every predicate in that cycle will be considered a subtype of every other, therefore rendering our sorting algorithm useless.

5.2.5 The other side of the coin: 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 following Racket forms, one for each of the three presented syntaxes:

- `(py-import <module-name> as <id>)`
- `(py-from <module-name> import ([<orig-id> as <bind-id>] ...)`
- `(py-from-import-* <module-name>)`

Above, `<module-name>` is a string containing the name of the module, while `<id>`, `<orig-id>` and `<bind-id>` are identifiers.

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 [12, ch. 4.17]. As for other Python operators, we provide the same Racket functions and macros we use on compiled code. These include:

- General language operators (attribute referencing, attribute setting, index referencing, ...) are provided as `py-get-attr`, `py-set-attr`, `py-get-index`, etc.

- Arithmetic binary operators (`+`, `-`, `*`, ...) are provided as `py-add`, `py-sub`, `py-mul`, etc.
- Comparisons (`<`, `>`, `==`, ...) are provided as `py-lt`, `py-gt`, `py-eq`, etc.
- Boolean operators (`or`, `and` and `not`) are provided as `py-or`, `py-and` and `py-not`, etc.

The end of the following section will contain an example detailing the use of some of these operators.

5.3 Interoperability 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.

5.3.1 Main strategy

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 with a slight change to the syntax described above for explicitly importing modules from CPython: replacing "`import`" with "`cpyimport`"

The module objects themselves are imported from the Python/C API using `PyImport_Import` [41]. 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.

In order to convert a module object from its CPython representation to our runtime representation, we must convert the module's name (a string) and the module's dictionary (whose keys are also strings, but whose values may be of any type). This entails being able to convert any Python object.

We achieved this flexibility by defining these two general-purpose functions:

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

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

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

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

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

5.3.2 Converting basic types

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

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

The Python/C API provides functions to convert these basic types to and from a C representation, while the Racket FFI handles the conversion from C to Racket. For example, converting an integer from CPython to Racket is as simple as:

```
(define (int-from-cpy x)
  (PyInt_AsLong x))
```

5.3.3 Converting type-objects

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

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

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

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

5.3.4 Converting opaque objects

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

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

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

In order for proxy objects to be applied as Racket procedures, we take advantage of the fact that structures are applicable in Racket. To this end, we define the following `prop:procedure` structure property [12, ch. 4.17]:

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

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

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

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

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

The semantics on exception handling is also well integrated with our implementation. Since we

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

5.3.5 Dealing with heterogeneity

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

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

```
1  log = []
2
3  def foo(n):
4    log.append(n)
5    return n * n
```

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

Suppose that we import and convert `log` to our list representation. When we call the `foo` proxy object, it updates the `log` list in CPython's shared memory, and not our converted list.

This issue is also present the other way around. Consider this example of dynamically adding a directory to `sys.path`, shown before, but this time for importing a module through CPython.

```
1  cpyimport sys
2  sys.path.append("/path/to/module/")
3  cpyimport foo
```

The `sys` module exports the `path` list. If we were to convert it to our list representation, appending a new path would only update the Racket list and not the one on CPython's shared memory. Therefore, we would not be able to import modules from other directories with CPython. The only way to keep track of the changes to `log` and `path` is by accessing their original C pointers via a proxy object.

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

We try to correct this by giving the user the power to explicitly convert proxy object collections to their standard representation. Python type-objects, when called, generally act as constructors for converting or copying objects from other types to that type. For instance, consider a tuple `a` and a list `b`.

The expression `list(a)` returns a new list with the contents of tuple `a`, while `list(b)` effectively returns a shallow copy of list `b`.

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

5.3.6 Implementing the `cpyimport` syntaxes

As mentioned at the beginning of the section, to use this import mechanism with `#lang python`, one simply has to replace the `import` keyword with `cpyimport`. Therefore, the supported syntaxes are:

1. `"cpyimport" <module> ["as" <name>]`
2. `"from" <module> "cpyimport" (<id> ["as" <name>])+`
3. `"from" <module> "cpyimport" "*"`

All three syntaxes make use of `PyImport_Import`, a Python/C API function which initializes and returns a module object given the module's name as a string.

The first one is implemented by converting the returned module object to our Racket our runtime representation, as described earlier, i.e. converting the module's name and its dictionary. This is done fully at runtime.

The second one is similar, but instead of converting every entry on the returned module object, we only convert the specified one(s).

As for the third syntax, just like before, we must define new bindings which are originally unknown at macro-expansion time. Unlike with Racket modules, whose initialization can be done in distinct and well defined phases, CPython can only provide us with the names of the imported definitions after initialising the module.

Therefore, we use `PyImport_Import` at macro-expansion time to initialize the module and obtain the names of the bindings we must define. This means that if the initialization of said module is supposed to cause any side-effects (e.g. printing a value, writing to a file, send packets through a network), these side-effects will take place during macro-expansion time, i.e. before the code has started running.

While this may result in unpredictable bugs due to anachronic side-effects, the only other alternatives would be replacing definitions with dynamic changes to the current namespace, therefore losing identifier traceability (as described in section 5.1), or not supporting this syntax at all.

Despite this pitfall, our main rationale for still supporting this syntax lies in the way it is typically used. Since `from...import*` imports everything from a module with no degree of control and is liable to shadow other imported or defined bindings, it is considered bad practice to use it on source files. Thus, its use in Python is mostly reserved for interactive REPL sessions. Since every iteration in Racket's read-eval-print-loop goes through a macro-expansion phase and runtime phase, any eventual side-effects resulting from the use of `from...cpyimport*` on the REPL will follow a correct order.

5.3.7 Using Python libraries in Racket

Once again, we have been describing how these imported libraries interoperate with our Python implementation, however, as with the standard `import` statements, this mechanism can also be accessed from Racket, by simply requiring the `python` language:

```
> (require python)
```

The `cpyimport...` and `from...cpyimport...` syntaxes are implemented by the `cpy-import` and `cpy-from` macros, respectively. Let us import `date` from Python's `datetime` module [40, ch. 8.1].

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

The identifier `date` is now available as a type-object. Let us get today's date (for the sake of example, suppose that today is the 14th of August 2014). This is done by getting the `today` function from the `date` type-object and calling it without arguments.

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 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.

This chapter will describe the effort needed in order to adapt DrRacket for Python development.

6.1 Source-code location

In chapter 4, we briefly mentioned the use of Racket's syntax-objects as the end result of the compilation process. As mentioned, a syntax-object for a particular piece of code contains that code's s-expression, source location information and lexical-binding information.

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.

Keep in mind that syntax-objects are composable, i.e. when using pre-existing syntax-objects to generate another one, each pre-existing syntax-object's properties are kept within the newly generated one. Therefore, by keeping the original source location information on every syntax object generated by the compiler, DrRacket can track each compiled s-expression to the corresponding Python code which produced it.

This way, DrRacket's features which rely on presenting info related to the source-code will work for Python. For instance, when editing Python source code, DrRacket will regularly parse and compile said code to Racket code and apply Racket's syntax checking. If the syntax checker detects a syntax error in the user's Python source code, this error will be signalled next to the piece of Python source-code which caused it, since that is the location that is featured on the syntax-object (**Fig. 6.1**).

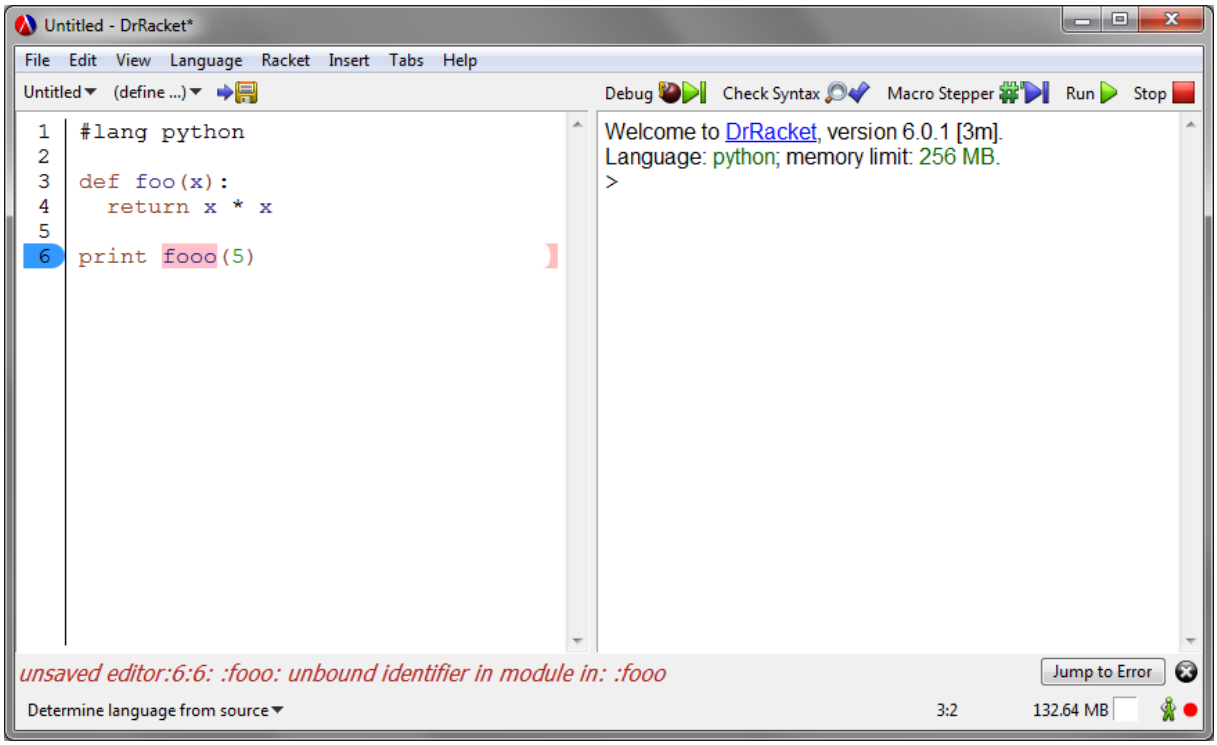


Figure 6.1: DrRacket displaying a syntax error

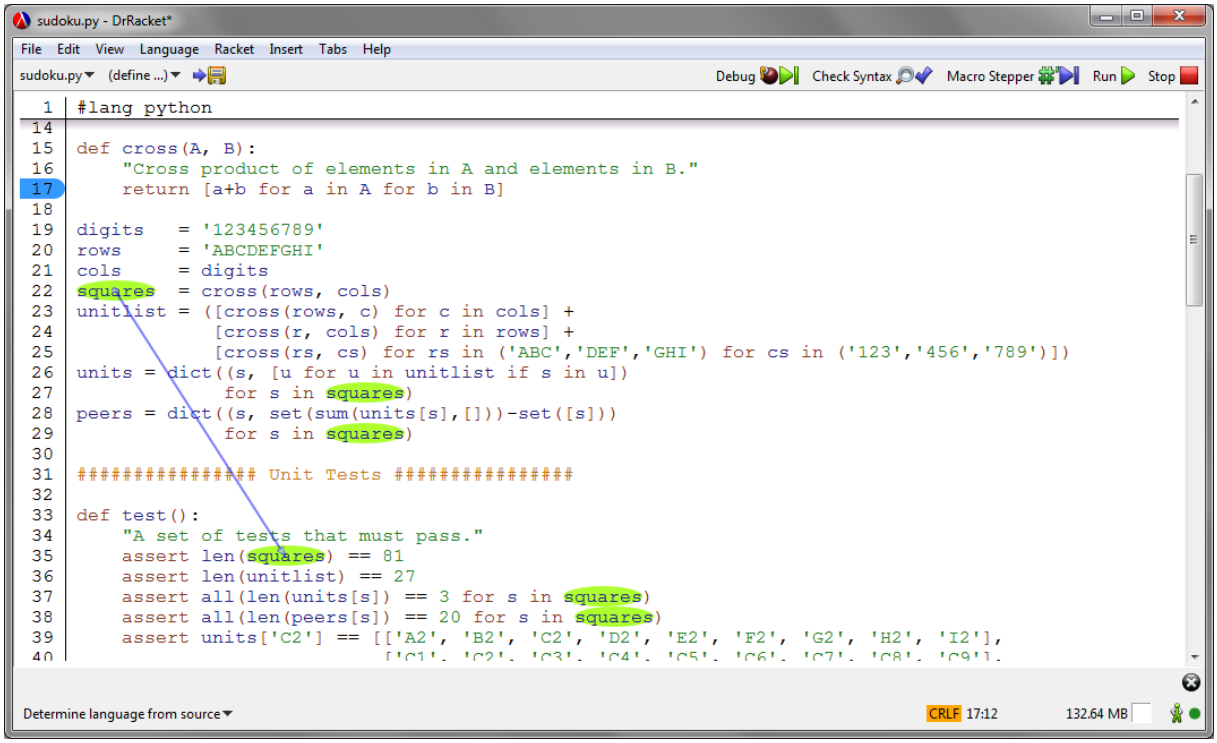


Figure 6.2: DrRacket tracking the definition of the `squares` variable

Another interesting feature which resulted from this is the tracking of bound identifiers. Racket can map any bound identifier to its definition, i.e. a `define` form, `let` form or `lambda` parameters. DrRacket takes advantage of this by pointing an arrow from the bound identifier to its binding definition, when the user hovers the mouse over either of them (**Fig. 6.2**). Since we compile the assignment of unbound identifiers to `define` forms, our Python identifiers can also be tracked within DrRacket. This is particularly useful in Python, for instance, to understand identifier scoping, since Python's scoping rules may not be as straightforward for a beginner as Racket's.

6.2 Syntax highlighting

DrRacket supports syntax highlighting for Racket code. Python code is evidently very different from Racket code in terms of syntax, thus we had to implement other rules for syntax highlighting.

Racket allows `#lang` developers to configure a language with arbitrary properties to be used by external tools. This is done by defining a `language-info` function that takes the property's symbol as input and returns its corresponding value. By defining a `language-info` for `#lang python`, these properties will not affect other languages such as `#lang racket`.

For syntax highlighting, DrRacket uses the `'color-lexer` property, which must map to a function that consumes text from an input port and returns tokens, i.e. a lexer. These tokens must refer to values from a color scheme (configurable from DrRacket's GUI menus).

Therefore, 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.

This results in a more comfortable coding experience for PyonR users with DrRacket, since it is now easier to distinguish elements such as language keywords from standard identifiers, comments, literals, among others. Using Racket's color scheme results in a familiar experience for Racket programmers since analogous elements are colored similarly. Additionally, this color scheme is fully customizable from DrRacket's Preferences menu (**Fig. 6.3**).

6.3 Read-Eval-Print-Loop (REPL)

DrRacket's REPL behaviour was also changed in order to be practical for a Python use. When a user presses the Enter key on the REPL, DrRacket looks for the `drracket:submit-predicate` property on `language-info`, which must map to a predicate that reads the input port and returns true if the inputted code is ready to be submitted to the `read` function. Otherwise, a newline character is entered and the user may continue typing.

Racket's submit predicate is designed for a Lisp syntax, returning false only when there are still parenthesis to close.

We implemented our own submit predicate by using the lexical analysis lexer to transform the input into tokens and analysing them. Python's syntax is influenced by indentation, so we return false if the

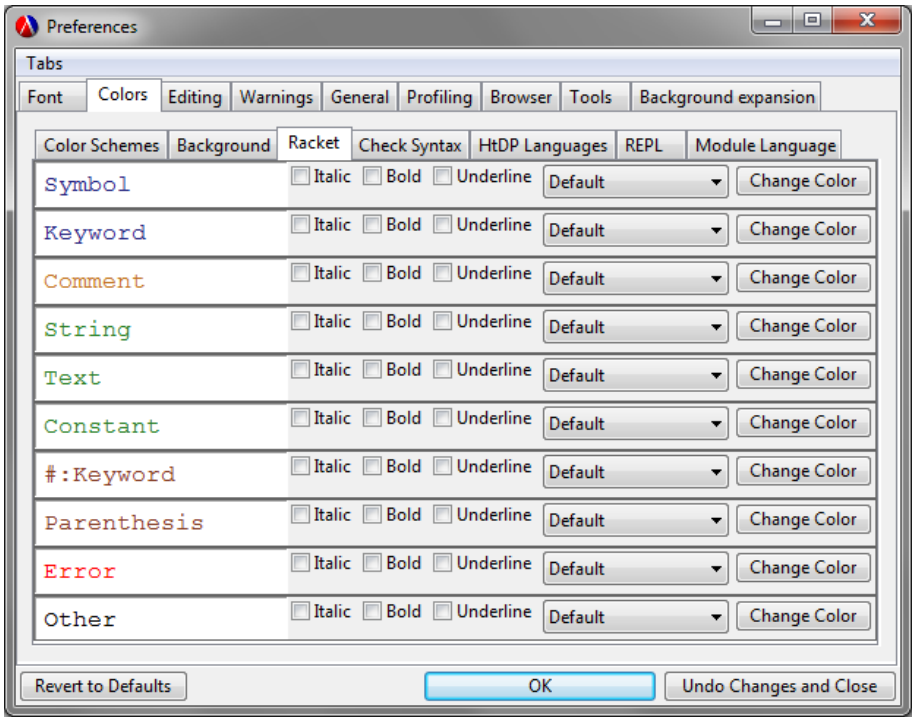


Figure 6.3: DrRacket's color scheme customization screen

number of dedents is less than the number of indents. Additionally, we also return false if the last token is a colon (meaning that the next group of substatements are part of the same statement) or if it is a backslash (signalling that the next newline is to be ignored).

In addition to the read phase, we also made changes to the print phase, otherwise Python values would be printed according to their Racket representation. This was done by dynamically changing Racket's `current-print` parameter. We configured it to call the `__repr__` method from the value's type-object and print it, therefore respecting Python's semantics.

6.4 Error Display Handler

As with printed values, the way unhandled Python exceptions are displayed should also be changed. This is especially important since our Python exception objects are not technically Racket exceptions, therefore DrRacket doesn't display their continuation marks (i.e. the stack trace) by default.

We did this also by dynamically changing a Racket parameter: `error-display-handler`. We defined it in terms of the original error display handler by simply building and displaying a Racket error value with a formatted error message according to Python's semantics and storing our continuation marks. This is demonstrated in **Fig. 6.4**.

Displaying the continuation marks works well for most cases, but there is still a slight issue when the unhandled exception was caused by a chain containing functions with tail calls. Racket, as a descent of Scheme, performs tail call optimization: when the last action in a function f is to call a function g, there is no need to store another stack frame for g on top of f because there is no need to return to f. Instead, f's stack frame is reused for g. This increases speed and, most importantly, avoids exhausting the stack.

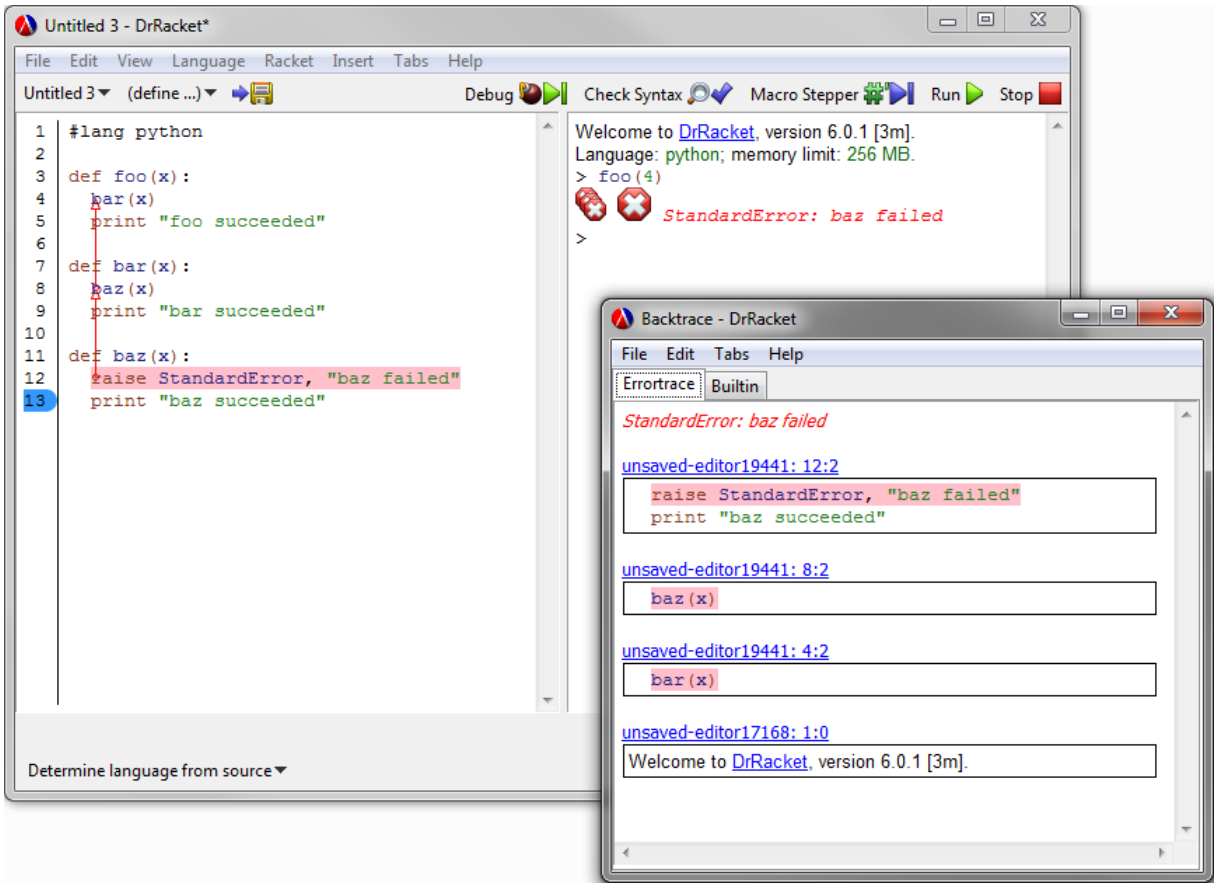


Figure 6.4: DrRacket displaying an exception's stacktrace alongside the code and in a separate window

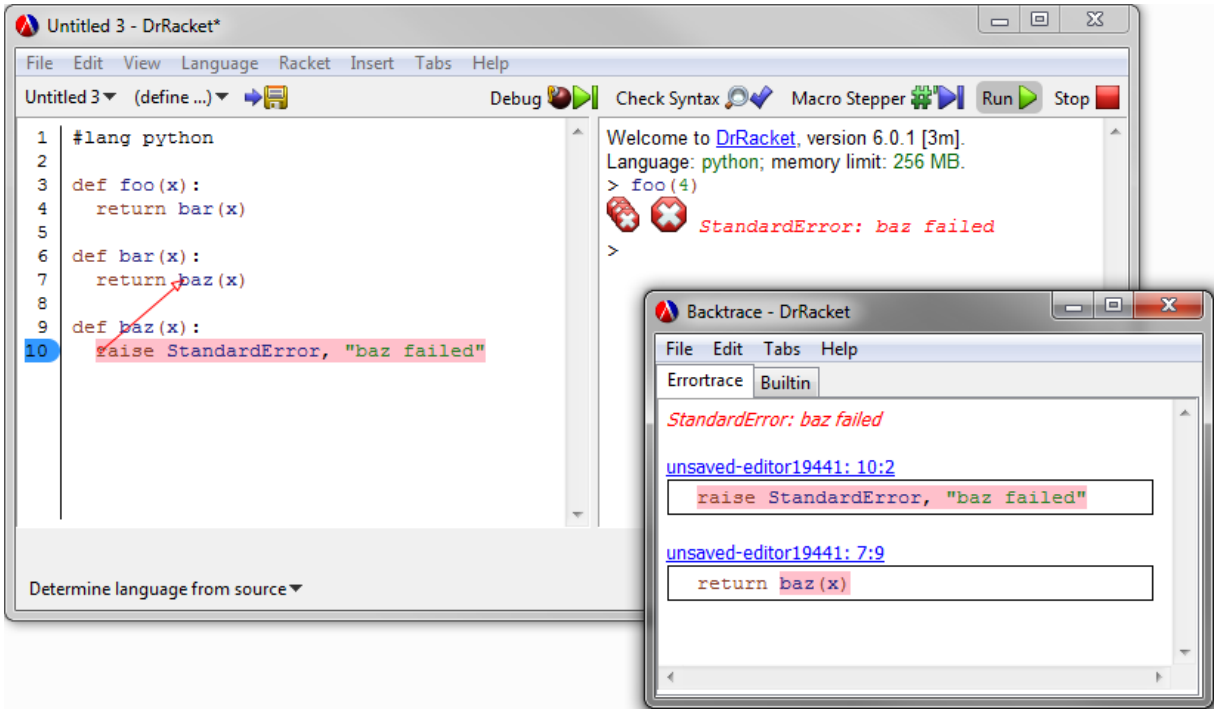


Figure 6.5: DrRacket displaying an exception's stacktrace with missing frames

However, tail call optimization turns out to be a double-edged sword for displaying stacktraces. Since stack frames are reused, the stacktrace will be missing some frames, which may make it more difficult for the user to track the origin of the error (**Fig. 6.5**).

This can be trivially solved by compiling function definitions in a way that avoids tall-calls (e.g. compiling the body of the function inside a `let` form and returning the bound value). We provide this modified function definition primitive in a submodule of `#lang python` named `debug`. This way, a user can simply change `#lang python` to `#lang python/debug` if he wishes to avoid tail-calls at all.

6.5 Language Customization

The Racket language provides a number of options which can be configured from DrRacket's Language Menu (**Fig. 6.6**). Since our `#lang python` is implemented on top of Racket, these options are also available to Python.

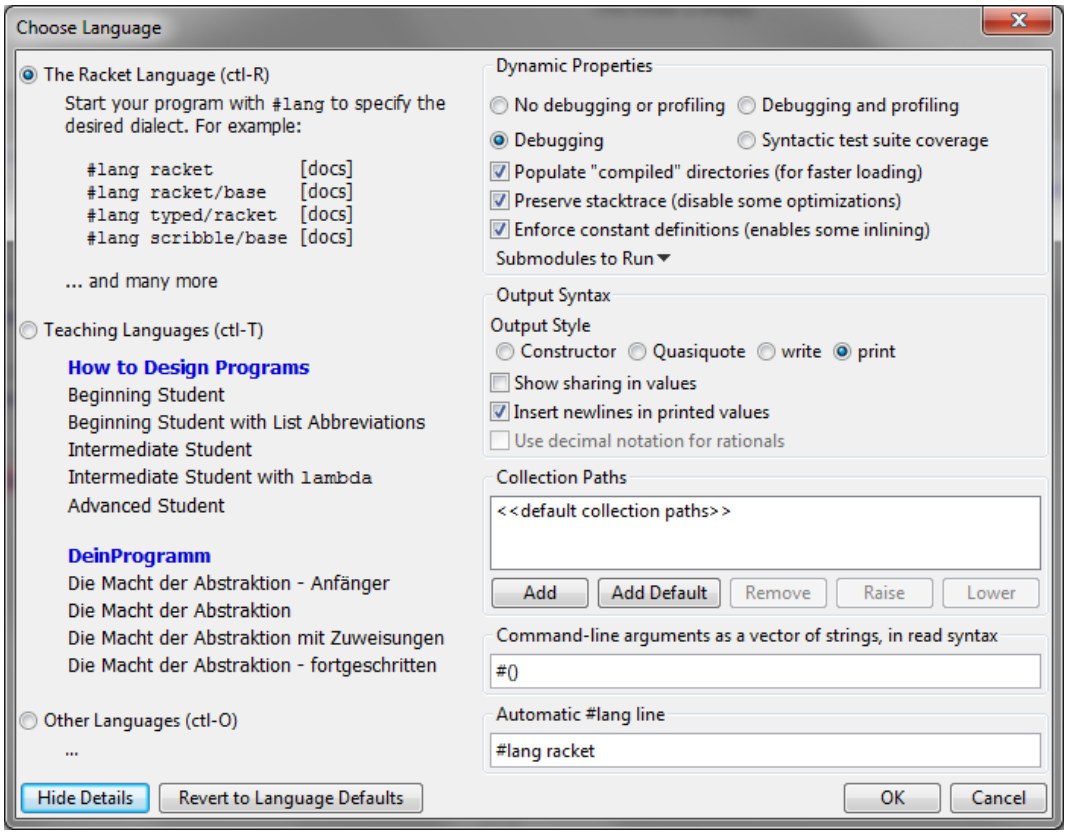


Figure 6.6: DrRacket's Language Configuration menu

6.5.1 Constant inlining

By default, Racket performs constant inlining at compile time when it detects that a variable is not reassigned within its module or by any of the functions defined inside the module. Just like with tail call optimization above, this is rarely an issue, but it can be when reassigning variables through meta-circular evaluators such as Racket's `eval` or, in our case, Python's `exec`.

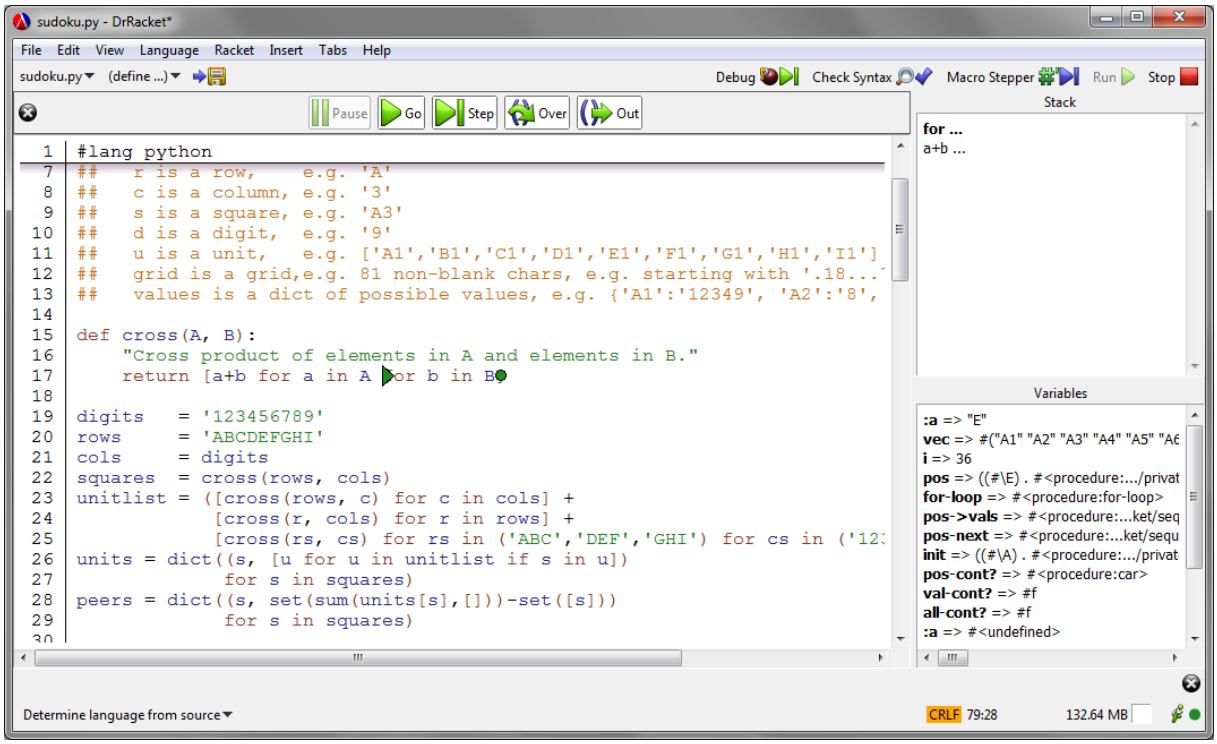


Figure 6.7: A DrRacket debugging session with Python

Consider the following program, where Racket's compiler erroneously judges variable a as a constant because its reassignment is not detected at compile time.

```
1 #lang python
2 a = 4
3 exec "a = 5"
4 print a
```

This results in a runtime error for attempting to modify a constant. However, after disabling constant inlining in DrRacket's Language menu, this becomes a valid program, printing the value 5.

6.5.2 Debugging and profiling

DrRacket features a step-by-step debugger and a profiler, both of which can be turned on or off in the Language menu.

Since we keep the source-code locations of the original Python code (as explained in section 6.1), users are able to track a debugging session along the source code (**Fig. 6.7**).

Stepping through Python statements sometimes requires pressing the Step button more than once, due to the overhead from compiling some Python statements to a sequence of Racket expressions, but otherwise the control flow followed by the debugger is predictable, which leads to a rather comfortable debugging experience.

As for the Variables pane, these will sometimes list variables which are only present on the compiled code (for instance, to enforce a control flow statement) and therefore are not present on the original Python code, but this also happens with Racket code, due to the use of macros.

As for the profiler, while it is possible to use it with Python, its results are not presented in a particularly readable way, as they will also reference function names from the runtime environment which are present on the compiled code.

Chapter 7

Performance

In this chapter we will showcase the current performance of our implementation versus CPython or Racket equivalents, as well as the performance gains from each one of the optimizations mentioned on section 3.4.

We will be presenting 5 examples, each one stressing different features of the Python language:

- Ackermann function – recursive function calls and arithmetic operators
- Mandelbrot function – `for` loop and early returns
- Mandelbrot function using classes – attribute getting and setting
- NumPy matrix addition – `cpyimport` type conversions and FFI calls
- Pystone benchmark – general Python benchmark

These benchmarks were performed on an Intel Core i7-3630QM processor at 3.2GHz, running Windows 7. The times below represent the minimum out of 5 samples.

Please note that we still haven't endeavoured into any serious attempt to profile these benchmarks, therefore these results only take into account the general optimizations described on section 3.4.

7.1 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:

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

print ackermann(3,9)
```

The chart on **Fig. 7.1** compares the running time of these examples for:

- **Racket** – Racket code running on Racket.
- **CPython** – Python code running on CPython.
- **PyonR (a)** – Python code running on Racket without early dispatch.
- **PyonR (b)** – Python code running on Racket with early dispatch (as explained on section 3.4.1).

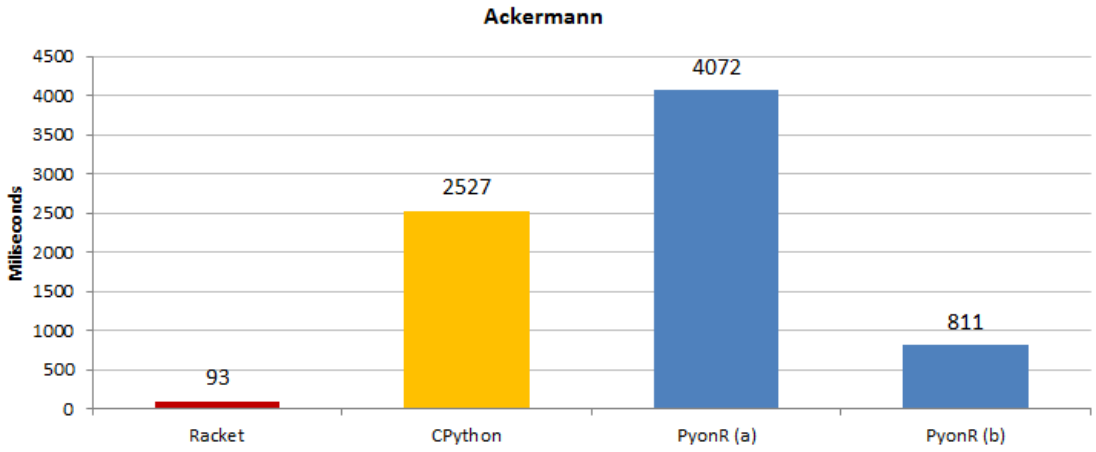


Figure 7.1: Benchmark of the Ackermann function

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.

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

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

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

Its Python equivalent could be implemented like such:

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

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

The chart on **Fig. 7.2** compares the running time of these examples for:

- **Racket** – Racket code running on Racket.
- **CPython** – Python code running on CPython.
- **PyonR (a)** – Python code running on Racket before optimizing sequence iteration.
- **PyonR (b)** – Python code running on Racket after optimizing sequence iteration (as explained on section 3.4.2).

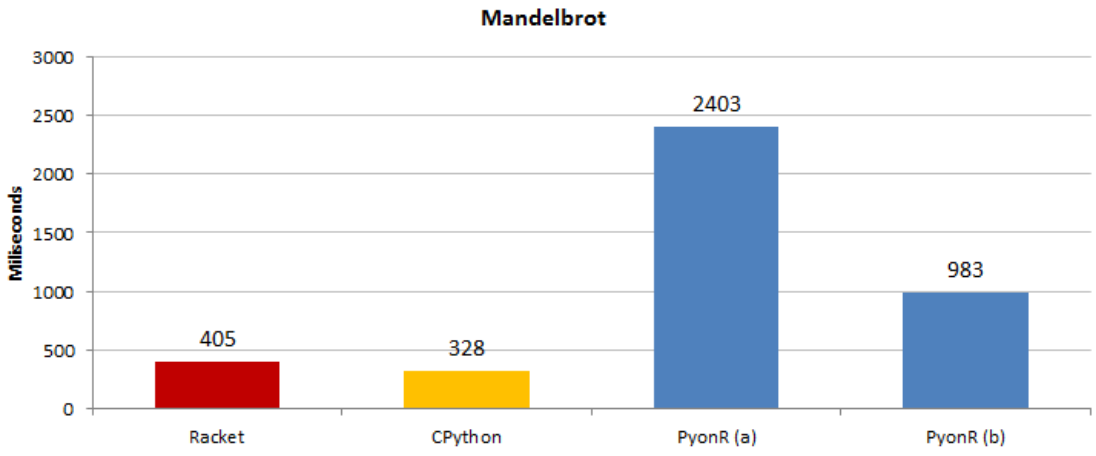


Figure 7.2: Benchmark of the Mandelbrot function

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:

```
1  class MandelbrotComplex:
2      def __init__(self):
3          self.z = 0+0j
4
5  def mandelbrot(limit, c):
6      mc = MandelbrotComplex()
7      for i in range(limit+1):
8          if abs(mc.z) > 2:
9              return i
10         mc.z = mc.z * mc.z + c
11     return i+1
12
13 print mandelbrot(1000000, .2+.3j)
```

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. 7.3** compares the running time of this example for:

- **CPython** – Python code running on CPython.
- **PyonR (a)** – Python code running on Racket before optimizing attribute getting and setting.
- **PyonR (b)** – Python code running on Racket after optimizing attribute getting and setting (as explained on section 3.4.3).

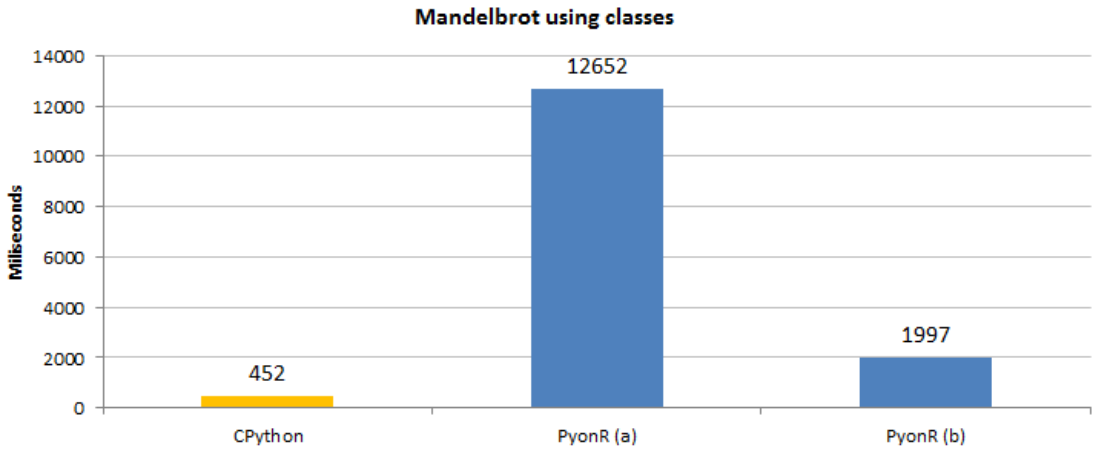


Figure 7.3: Benchmark of the Mandelbrot function using classes

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 NumPy Matrix Addition

The following benchmark aims to measure the overhead introduced by our type conversions and foreign function class when using imported libraries from CPython with the `cpyimport` statement. Consider the Python example below, using the NumPy library, where we define and call a function which adds a given number of 100×100 matrices with random integers up to 100000.

```
1  import numpy as np
2
3  def add_arrays(n):
4    result = np.zeros((100,100))
5    for i in range(n):
6      result += np.random.randint(0, 100000, (100,100))
7    return result
8
9  print add_arrays(10000)
```

To get this code running with PyonR, we simply have to declare its language with `#lang python` as the first line and replace `import` with `cpyimport`.

Consider also a second version where we define the `add_arrays` function in a separate file named `arrays_example.py` and accessible from CPython's `sys.path` and we import it and call it with PyonR, like this:

```
1  #lang python
2  from arrays_example cpyimport add_arrays
3
4  print add_arrays(10000)
```

The chart on **Fig. 7.4** compares the running time of this example for:

- **CPython** – Original Python code running on CPython.
- **PyonR (a)** – Original Python code running on PyonR.
- **PyonR (b)** – Single call version running on PyonR.

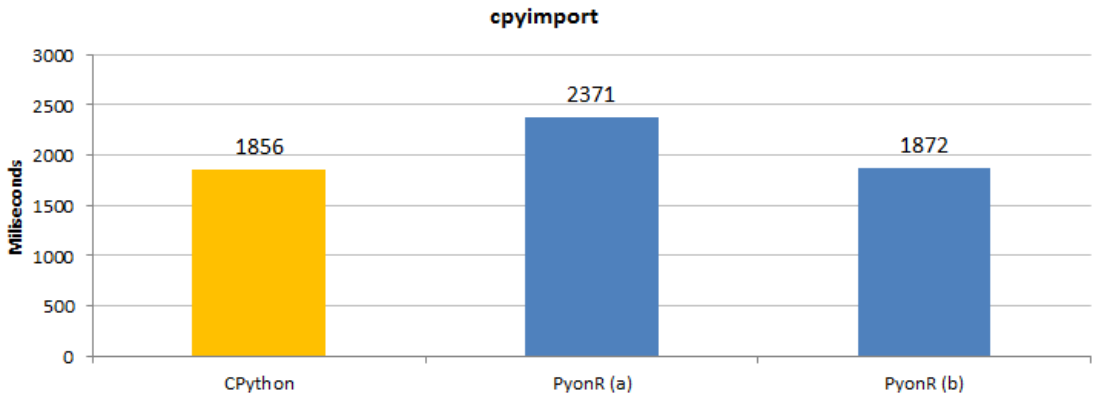


Figure 7.4: Benchmark of the NumPy example, using `cpyimport`

Using PyonR, we get a running time of $2371ms$, which is around 30% slower than using CPython directly. This is a very acceptable overhead for most use cases, but it may be an issue for high-performance

applications.

Fortunately, the single call version demonstrates an alternative which virtually eliminates the overhead from FFI calls and type conversions. This time, instead of dealing with FFI calls and type conversions on every iteration of the `for` cycle, we simply have one foreign function call to deal with, since the computation of `add_arrays` is handled by CPython. The measured running time for this example is $1872ms$, which is nearly identical to the one measured for CPython.

7.5 Pystone

We conclude this chapter with the Pystone benchmark [20]. 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. Its implementation is provided in the Appendix.

The chart on **Fig. 7.5** compares the running time of this example for:

- **CPython** – Pystone running on CPython.
- **PyonR (a)** – Pystone running on PyonR without the described optimizations.
- **PyonR (b)** – Pystone running on PyonR with early dispatch.
- **PyonR (c)** – Pystone running on PyonR with early dispatch and sequence iteration optimization.
- **PyonR (d)** – Pystone running on PyonR with early dispatch, sequence iteration optimization, and attribute getting and setting optimizations.

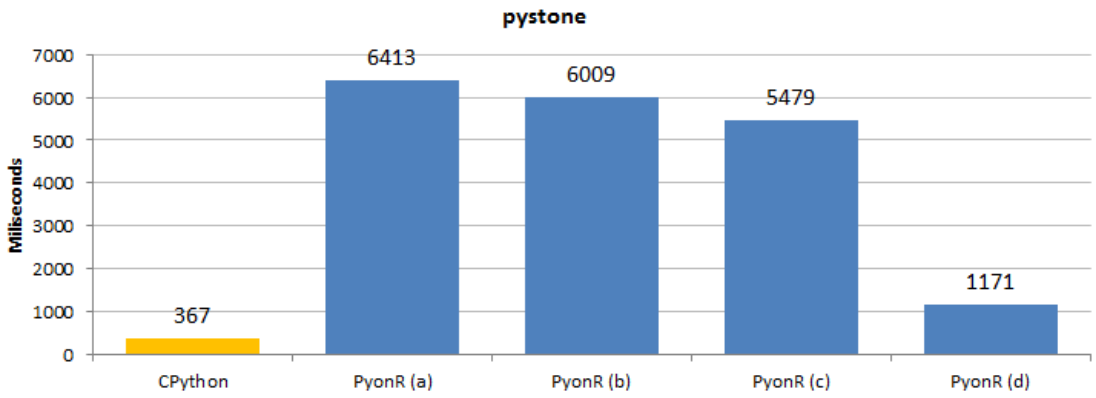


Figure 7.5: Pystone benchmark

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.

Chapter 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. This runtime environment is able to natively interoperate with Racket libraries without any additional performance overheads and also with libraries imported from Python's reference implementation with a small performance overhead. Additionally, this runtime environment can be used from the Racket language directly, providing the ability to embed Python functionality in Racket code.

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

- **Correctness and completeness** – PyonR currently implements most of the Python language (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 `cpyimport` 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 chapter 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 chapter 6.

- **Interoperability with Racket** – we extended Python's `import` statement syntax 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.

On chapter 2, we have compared other Python implementations with Table 2.1. We can now include PyonR in that comparison with Table 8.1.

	Language(s) written	**Platform(s) targeted**	**Speedup (vs. CPython)**	**Std. library support**
CPython (1994-)	C	CPython's VM	$1\times$	Full
Jython (2000-)	Java	JVM	$\sim 1\times$	Most
IronPython (2006-)	C#	CLI	$\sim 1.8\times$	Most
PyPy (2007-)	RPython	C, JVM, CLI	$\sim 6\times$	Most
CLPython (2006-2013)	Common Lisp	Common Lisp	$\sim 0.5\times$	Most
PLT Spy (2003-2005)	PLT Scheme, C	PLT Scheme	$\sim 0.001\times$	Full
PyonR (2014-)	Racket	Racket	$\sim 0.3\times$	Full

Table 8.1: Comparison between implementations, including PyonR

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.

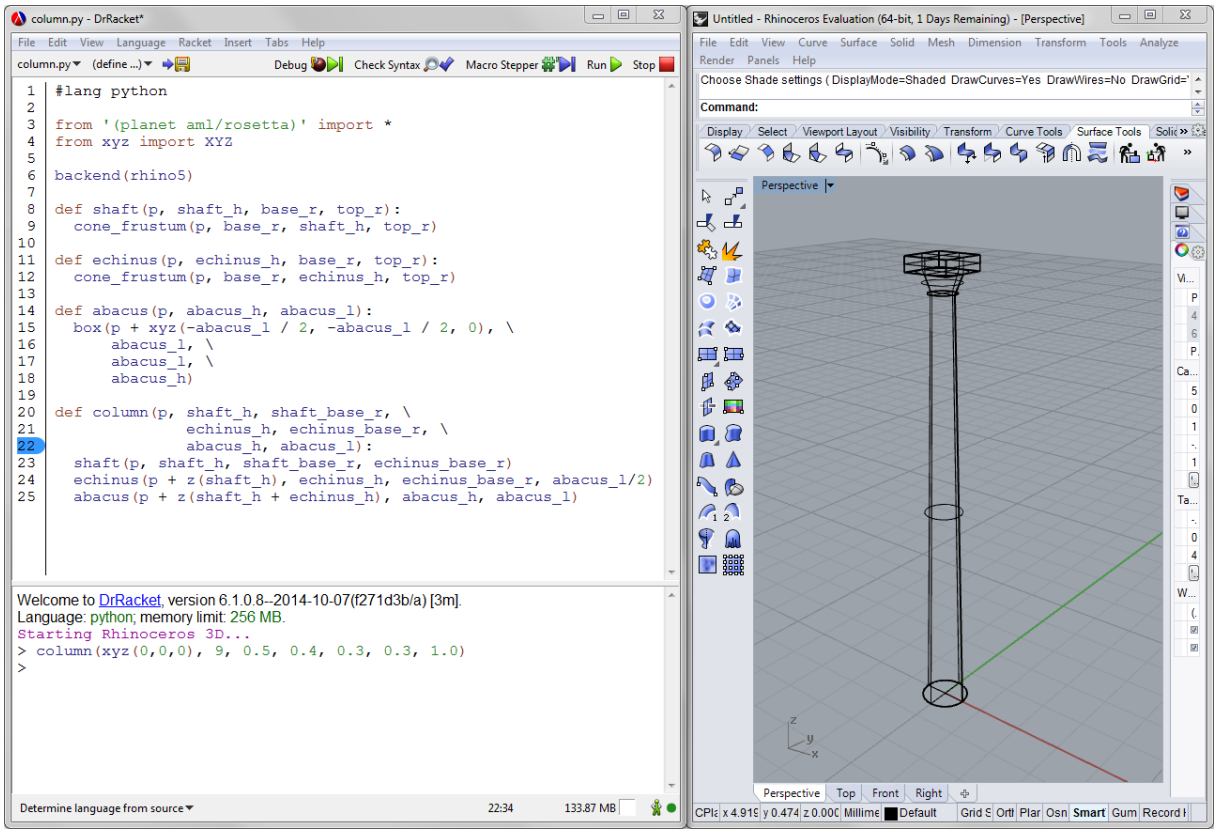


Figure 8.1: Rosetta being used with the Python language as front-end and Rhinoceros as back-end. Rosetta's features are imported from PLaneT with our modified `import` syntax. This allows us to select a backend, like before, and access every other provided feature.

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 (**Fig. 8.1**).

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 in order to speed up PyonR. Such optimizations could include:

- Rewriting the control flow for loops 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. For instance, a Python statement like:

```
for i in range(1000):
    <body>
```

can be compiled to:

```
(for ([i (in-range 1000)])
  <body>)
```

which avoids the overhead of building a Python list and yields a much faster iteration;

- Rewriting parts of the runtime environment in Typed Racket to avoid the performance overhead of checking types at runtime;
- Profiling Python examples to find and optimize the implementation's weak-points.

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.

Bibliography

[1] Kim Barrett, Bob Cassels, Paul Haahr, David A Moon, Keith Playford, and P Tucker Withington. A monotonic superclass linearization for Dylan. In *ACM SIGPLAN Notices*, volume 31, pages 69–82. ACM, 1996.

[2] Eli Barzilay. *The Racket Foreign Interface*, 2012.

[3] David Beazley. Understanding the Python GIL. In *PyCON Python Conference*, Atlanta, Georgia, 2010.

[4] David M Beazley. *Python Essential Reference*. Addison-Wesley Professional, 2009.

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

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

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

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

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

[10] M Flatt and RB Findler. *The Racket Guide*, 2013.

[11] Matthew Flatt. *PLT Scheme C API*.

[12] Matthew Flatt et al. *The Racket Reference*. PLT, 2013.

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

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

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

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

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

[18] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator, 1975.

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

[20] Mark Lutz. *Learning python*, chapter "Other Benchmarking Topics: pystones". " O'Reilly Media, Inc.", 2013.

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

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

[23] Peter Norvig. Python for Lisp programmers. `http://norvig.com/python-lisp.html`. [Online; retrieved on October 2014].

[24] Scott Owens. *Parser Tools:* `lex` *and* `yacc`*-style Parsing*, 2012.

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

[26] Benjamin Peterson. PEP 373 – Python 2.7 release schedule. `http://legacy.python.org/dev/peps/pep-0373/`, July 2014.

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

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

[29] Pedro Palma Ramos and António Menezes Leitão. An implementation of Python for Racket. In *7th European Lisp Symposium*, page 72, 2014.

[30] Pedro Palma Ramos and António Menezes Leitão. Implementing Python for DrRacket. In *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 127–141, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[31] Pedro Palma Ramos and António Menezes Leitão. Reaching Python from Racket. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, page 32. ACM, 2014.

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

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

[34] G Steele Jr. Lambda: The ultimate goto. Technical report, AI Lab Memo, 1977.

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

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

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

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

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

[40] Guido van Rossum and Fred L Drake. *Python Library Reference*, 2014.

[41] Guido van Rossum and Fred L Drake Jr. *Python/C API Reference Manual*, 2002.

Appendix

Pystone Benchmark

```
#lang python

"""
"PYSTONE" Benchmark Program

Version:        Python/1.1 (corresponds to C/1.1 plus 2 Pystone fixes)

Author:         Reinhold P. Weicker,  CACM Vol 27, No 10, 10/84 pg. 1013.

                Translated from ADA to C by Rick Richardson.
                Every method to preserve ADA-likeness has been used,
                at the expense of C-ness.

                Translated from C to Python by Guido van Rossum.

Version History:

                Version 1.1 corrects two bugs in version 1.0:

                First, it leaked memory: in Proc1(), NextRecord ends
                up having a pointer to itself.  I have corrected this
                by zapping NextRecord.PtrComp at the end of Proc1().

                Second, Proc3() used the operator != to compare a
                record to None.  This is rather inefficient and not
                true to the intention of the original benchmark (where
                a pointer comparison to None is intended; the !=
                operator attempts to find a method __cmp__ to do value
                comparison of the record).  Version 1.1 runs 5-10
                percent faster than version 1.0, so benchmark figures
                of different versions can't be compared directly.

"""

LOOPS = 50000

from time cpyimport clock

__version__ = "1.1"

[Ident1, Ident2, Ident3, Ident4, Ident5] = range(1, 6)

```

```
43  class Record:
44
45      def __init__(self, PtrComp = None, Discr = 0, EnumComp = 0,
46                           IntComp = 0, StringComp = 0):
47          self.PtrComp = PtrComp
48          self.Discr = Discr
49          self.EnumComp = EnumComp
50          self.IntComp = IntComp
51          self.StringComp = StringComp
52
53      def copy(self):
54          return Record(self.PtrComp, self.Discr, self.EnumComp,
55                        self.IntComp, self.StringComp)
56
57  TRUE = 1
58  FALSE = 0
59
60  def main(loops=LOOPS):
61      benchtime, stones = pystones(loops)
62      print "Pystone(%s) time for %d passes = %g" % \
63            (__version__, loops, benchtime)
64      print "This machine benchmarks at %g pystones/second" % stones
65
66
67  def pystones(loops=LOOPS):
68      return Proc0(loops)
69
70
71  IntGlob = 0
72  BoolGlob = FALSE
73  Char1Glob = '\0'
74  Char2Glob = '\0'
75  Array1Glob = [0]*51
76  Array2Glob = map(lambda x: x[:], [Array1Glob]*51)
77  PtrGlb = None
78  PtrGlbNext = None
79
80  def Proc0(loops=LOOPS):
81      global IntGlob
82      global BoolGlob
83      global Char1Glob
84      global Char2Glob
85      global Array1Glob
86      global Array2Glob
87      global PtrGlb
88      global PtrGlbNext
89
90      starttime = clock()
91      for i in range(loops):
92          pass
93      nulltime = clock() - starttime
94
95      PtrGlbNext = Record()
96      PtrGlb = Record()
97      PtrGlb.PtrComp = PtrGlbNext
98      PtrGlb.Discr = Ident1
99      PtrGlb.EnumComp = Ident3
100     PtrGlb.IntComp = 40
```

```
    PtrGlb.StringComp = "DHRYSTONE PROGRAM, SOME STRING"
    String1Loc = "DHRYSTONE PROGRAM, 1'ST STRING"
    Array2Glob[8][7] = 10

    starttime = clock()

    for i in range(loops):
        Proc5()
        Proc4()
        IntLoc1 = 2
        IntLoc2 = 3
        String2Loc = "DHRYSTONE PROGRAM, 2'ND STRING"
        EnumLoc = Ident2
        BoolGlob = not Func2(String1Loc, String2Loc)
        while IntLoc1 < IntLoc2:
            IntLoc3 = 5 * IntLoc1 - IntLoc2
            IntLoc3 = Proc7(IntLoc1, IntLoc2)
            IntLoc1 = IntLoc1 + 1
        Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3)
        PtrGlb = Proc1(PtrGlb)
        CharIndex = 'A'
        while CharIndex <= Char2Glob:
            if EnumLoc == Func1(CharIndex, 'C'):
                EnumLoc = Proc6(Ident1)
            CharIndex = chr(ord(CharIndex)+1)
        IntLoc3 = IntLoc2 * IntLoc1
        IntLoc2 = IntLoc3 / IntLoc1
        IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1
        IntLoc1 = Proc2(IntLoc1)

    benchtime = clock() - starttime - nulltime
    if benchtime == 0.0:
        loopsPerBenchtime = 0.0
    else:
        loopsPerBenchtime = (loops / benchtime)
    return benchtime, loopsPerBenchtime

def Proc1(PtrParIn):
    PtrParIn.PtrComp = NextRecord = PtrGlb.copy()
    PtrParIn.IntComp = 5
    NextRecord.IntComp = PtrParIn.IntComp
    NextRecord.PtrComp = PtrParIn.PtrComp
    NextRecord.PtrComp = Proc3(NextRecord.PtrComp)
    if NextRecord.Discr == Ident1:
        NextRecord.IntComp = 6
        NextRecord.EnumComp = Proc6(PtrParIn.EnumComp)
        NextRecord.PtrComp = PtrGlb.PtrComp
        NextRecord.IntComp = Proc7(NextRecord.IntComp, 10)
    else:
        PtrParIn = NextRecord.copy()
    NextRecord.PtrComp = None
    return PtrParIn

def Proc2(IntParIO):
    IntLoc = IntParIO + 10
    while 1:
        if Char1Glob == 'A':
            IntLoc = IntLoc - 1
```

```
159             IntParIO = IntLoc - IntGlob
160             EnumLoc = Ident1
161         if EnumLoc == Ident1:
162             break
163     return IntParIO
164
165 def Proc3(PtrParOut):
166     global IntGlob
167
168     if PtrGlb is not None:
169         PtrParOut = PtrGlb.PtrComp
170     else:
171         IntGlob = 100
172     PtrGlb.IntComp = Proc7(10, IntGlob)
173     return PtrParOut
174
175 def Proc4():
176     global Char2Glob
177
178     BoolLoc = Char1Glob == 'A'
179     BoolLoc = BoolLoc or BoolGlob
180     Char2Glob = 'B'
181
182 def Proc5():
183     global Char1Glob
184     global BoolGlob
185
186     Char1Glob = 'A'
187     BoolGlob = FALSE
188
189 def Proc6(EnumParIn):
190     EnumParOut = EnumParIn
191     if not Func3(EnumParIn):
192         EnumParOut = Ident4
193     if EnumParIn == Ident1:
194         EnumParOut = Ident1
195     elif EnumParIn == Ident2:
196         if IntGlob > 100:
197             EnumParOut = Ident1
198         else:
199             EnumParOut = Ident4
200     elif EnumParIn == Ident3:
201         EnumParOut = Ident2
202     elif EnumParIn == Ident4:
203         pass
204     elif EnumParIn == Ident5:
205         EnumParOut = Ident3
206     return EnumParOut
207
208 def Proc7(IntParI1, IntParI2):
209     IntLoc = IntParI1 + 2
210     IntParOut = IntParI2 + IntLoc
211     return IntParOut
212
213 def Proc8(Array1Par, Array2Par, IntParI1, IntParI2):
214     global IntGlob
215
216     IntLoc = IntParI1 + 5
```

```
    Array1Par[IntLoc] = IntParI2
    Array1Par[IntLoc+1] = Array1Par[IntLoc]
    Array1Par[IntLoc+30] = IntLoc
    for IntIndex in range(IntLoc, IntLoc+2):
        Array2Par[IntLoc][IntIndex] = IntLoc
    Array2Par[IntLoc][IntLoc-1] = Array2Par[IntLoc][IntLoc-1] + 1
    Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc]
    IntGlob = 5

def Func1(CharPar1, CharPar2):
    CharLoc1 = CharPar1
    CharLoc2 = CharLoc1
    if CharLoc2 != CharPar2:
        return Ident1
    else:
        return Ident2

def Func2(StrParI1, StrParI2):
    IntLoc = 1
    while IntLoc <= 1:
        if Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1:
            CharLoc = 'A'
            IntLoc = IntLoc + 1
    if CharLoc >= 'W' and CharLoc <= 'Z':
        IntLoc = 7
    if CharLoc == 'X':
        return TRUE
    else:
        if StrParI1 > StrParI2:
            IntLoc = IntLoc + 7
            return TRUE
        else:
            return FALSE

def Func3(EnumParIn):
    EnumLoc = EnumParIn
    if EnumLoc == Ident3: return TRUE
    return FALSE

if __name__ == '__main__':
    main(LOOPS)
```