An Implementation of Python for Racket

Pedro Palma Ramos
António Menezes Leitão
Contents

• Motivation
• Goals
• Related Work
• Solution
• Performance Benchmarks
• Future Work
Racket + DrRacket

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

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 256 MB.
>(ackermann 3 9)
4093
>

Determine language from source ▼
Racket + DrRacket

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

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 256 MB.
> (ackermann 3 9)
4093
>
Pedro Ramos & António Menezes Leitão | An Implementation of Python for Racket
Our goal...

```python
#lang python

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))
```

Welcome to DrRacket, version 5.3.6 [3m].
Language: python; memory limit: 256 MB.
> ackermann(3, 9)
4093
>
Our goal...
Why Python?

Python is replacing Scheme in introductory programming courses
#lang racket

(require rosetta)

(backend "rhino")

(define (randomize-sphere d1 d2 sr shape-fn)
  (let* ((r (random-interval d1 d2))
         (th (random-interval 0 (* pi 2)))
         (fi (random-interval 0 pi))
         (p (sph r th fi))
         (move p (shape-fn (- sr r) p))))

(define (shape-cloud d1 d2 r n shape-fn)
  (for/list ((i (range n))
             (randomize-sphere d1 d2 r shape-fn))
    (shape-cloud d1 d2 r n shape-fn))

(shape-cloud 4 5 5 600
  (λ (r n) (box-normal r r r n)))

#<rhino-shape>
#<rhino-shape>
#<rhino-shape>
#<rhino-shape>
#<rhino-shape>
#<rhino-shape>
Rosetta IDE

Front ends:

- JavaScript
- Racket
- AutoLISP
- Python

Back ends:

- AutoCAD
- Rhinoceros
- OpenGL
• Borrows influences from Lisp
• High level, dynamically typed, GC’d
• Multiple paradigms
• Huge standard library + third-party libraries
Goals

• Correctness + Completeness
• Performance
• DrRacket Integration
• Interoperability with Racket
## Related implementations

<table>
<thead>
<tr>
<th>Language(s) written</th>
<th>Platform(s) targetted</th>
<th>Speedup (vs. CPython)</th>
<th>Std. library support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython</td>
<td>C</td>
<td>CPython’s VM</td>
<td>1x</td>
</tr>
</tbody>
</table>

CPython is written in C, targeted to CPython’s VM, with a speedup of 1x compared to CPython, and full support for the standard library.
## Related implementations

<table>
<thead>
<tr>
<th>Language(s) written</th>
<th>Platform(s) targetted</th>
<th>Speedup (vs. CPython)</th>
<th>Std. library support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython</td>
<td>C</td>
<td>CPython’s VM</td>
<td>1x</td>
</tr>
<tr>
<td>Jython</td>
<td>Java</td>
<td>JVM</td>
<td>~1x</td>
</tr>
<tr>
<td>IronPython</td>
<td>C#</td>
<td>CLI</td>
<td>~1.8x</td>
</tr>
<tr>
<td>CLPython</td>
<td>Common Lisp</td>
<td>Common Lisp</td>
<td>~0.5x</td>
</tr>
</tbody>
</table>
## Related implementations

<table>
<thead>
<tr>
<th></th>
<th>Language(s) written</th>
<th>Platform(s) targeted</th>
<th>Speedup (vs. CPython)</th>
<th>Std. library support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython</td>
<td>C</td>
<td>CPython’s VM</td>
<td>1x</td>
<td>Full</td>
</tr>
<tr>
<td>Jython</td>
<td>Java</td>
<td>JVM</td>
<td>~1x</td>
<td>Most</td>
</tr>
<tr>
<td>IronPython</td>
<td>C#</td>
<td>CLI</td>
<td>~1.8x</td>
<td>Most</td>
</tr>
<tr>
<td>CLPython</td>
<td>Common Lisp</td>
<td>Common Lisp</td>
<td>~0.5x</td>
<td>Most</td>
</tr>
<tr>
<td>PLT Spy</td>
<td>Scheme, C</td>
<td>Scheme</td>
<td>~0.001x</td>
<td>Full</td>
</tr>
</tbody>
</table>
Our solution...
Pipeline

Python → Racket (source-to-source compiler)

Racket → Bytecode (Racket compiler + JIT)
Architecture

Legend:

A \rightarrow \text{uses} \rightarrow B

Pedro Ramos & António Menezes Leitão | An Implementation of Python for Racket
Racket Modules

• reader module (for compilation)
  – read: input-port → (listof s-expression)
  – read-syntax: input-port → (listof syntax-object)

• python module (for runtime behaviour)
  – Provides functions/macros used in compiled code
Syntax-objects

- S-expression
- Source location information
  - File, line number, column number, span
- Lexical-binding information
1 | #lang python
2 | arr = [1,1,2,3,5,8,13,21]
3 | print arr[6]

\[
(py\text{-}print\ (py\text{-}get\text{-}index\ arr\ 6))
\]
\[
\text{line: 3, cols: 0-12}
\]

py\text{-}print

\[
(py\text{-}get\text{-}index\ arr\ 6)
\]
\[
\text{line: 3, cols: 6-12}
\]

py\text{-}get\text{-}index

\[
arr
\]
\[
\text{line: 3, cols: 6-9}
\]

6
\[
\text{line: 3, cols: 10-11}
\]
Architecture

Legend:

A uses B
How to implement Python’s behaviour?
Runtime implementation

Two alternatives:

• Mapping to Python/C API
  (via Racket Foreign Function Interface)

• Racket reimplementation
FFI Approach

libpython module

Racket FFI
(Foreign Function Interface)

foreign calls on C pointers

Python/C API

Racket

CPython VM
(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 (range (length elems))]
          [elem elems])
      (PyTuple_SetItem py-tuple i elem)
      py-tuple))
)
FFI Runtime - Disadvantages

• Bad Performance
  – Expensive type conversions + FFI calls
  – Finalizers for GC

• Clumsy Interoperability with Racket
  – Wrappers/Unwrappers
What about implementing it over Racket data types?

We must first understand Python’s data model
Python’s Data Model

• Every value is an object

• Every object has a reference to its type-object

• Type-objects hold hash-table for method dispatching
  – Maps method names to function objects

• Operator behaviour is mapped to methods
Optimizations

• Basic types mapped to Racket types
  – int, long, float, complex, string, dict
  – Avoids wrapping/unwrapping

• Early method dispatching for operators
  – Avoids expensive method dispatching for common uses
(define (py-add x y)
  (py-method-call x "__add__" y))

(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)]))
How are modules imported?
Python import system

- `import <module>`
  - `<module>` is imported as a module object
- `from <module> import <id>`
  - `<id>` is imported as a new binding
- `from <module> import *`
  - All bindings from `<module>` are imported

- Special syntax for Racket imports
#lang python
import "racket" as rkt

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

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

> add_cons(c1)
5
> add_cons(c2)
"abcdef"
#lang python
from "racket" import cons, car, cdr

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

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

> add_cons(c1)
5
> add_cons(c2)
"abcdef"
#lang python
from "racket\trace" import trace

def factorial(n):
    if n == 0: return 1
    else: return n * factorial(n-1)

trace(factorial)

> factorial(5)
>(factorial 5)
>(factorial 4)
>(factorial 3)
>(factorial 2)
>(factorial 1)
>> (factorial 0)
<< 1
<<1
<< 2
<< 6
<< 24
<< 120
120
Other Features

• Class definitions
  – class statement $\rightarrow$ new type object

• Exception handling
  – raise, try...except statements $\rightarrow$ raise, with-handlers forms

• Flow control statements
  – return, break, continue, yield $\rightarrow$ escape continuations
Benchmarks

- Ackermann
  - computing the Ackermann function

- Mandelbrot
  - computing if a complex sequence diverges after a limited number of iterations
(define (ackermann m n)
  (cond
    [ (= m 0) (+ n 1)]
    [(and (> m 0) (= n 0)) (ackermann (- m 1) 1)]
    [else (ackermann (- m 1) (ackermann m (- n 1)))]))

(ackermann 3 9)

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)
Mandelbrot

(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)

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)
Ackermann - Results

<table>
<thead>
<tr>
<th></th>
<th>Miliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>Racket on Racket</td>
</tr>
<tr>
<td>(b)</td>
<td>Python on CPython</td>
</tr>
<tr>
<td>(c.1)</td>
<td>Python on Racket (FFI)</td>
</tr>
<tr>
<td>(c.2)</td>
<td>Python on Racket (FFI + finalizers)</td>
</tr>
<tr>
<td>(d.1)</td>
<td>Python on Racket (native)</td>
</tr>
<tr>
<td>(d.2)</td>
<td>Python on Racket (native + early dispatch)</td>
</tr>
</tbody>
</table>

- (a) Racket on Racket: 93 ms
- (b) Python on CPython: 2641 ms
- (c.1) Python on Racket (FFI): 54351 ms
- (c.2) Python on Racket (FFI + finalizers): 115557 ms
- (d.1) Python on Racket (native): 11357 ms
- (d.2) Python on Racket (native + early dispatch): 811 ms

Pedro Ramos & António Menezes Leitão | An Implementation of Python for Racket
Mandelbrot - Results

<table>
<thead>
<tr>
<th></th>
<th>Racket on Racket</th>
<th>Python on CPython</th>
<th>Python on Racket (FFI)</th>
<th>Python on Racket (FFI + finalizers)</th>
<th>Python on Racket (native)</th>
<th>Python on Racket (native + early dispatch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c.1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d.1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Miliseconds**

- (a): Racket on Racket
- (b): Python on CPython
- (c.1): Python on Racket (FFI)
- (c.2): Python on Racket (FFI + finalizers)
- (d.1): Python on Racket (native)
- (d.2): Python on Racket (native + early dispatch)
Future Work

• Fully implement compilation process
• Implement behaviour for built-in types
• Integrate FFI calls with current data model
• Formal testing for correctness
Thank you for listening!

Questions? Comments?