RPython
A Step Towards Reconciling
Dynamically and Statically Typed OO Languages

Antonio Cuni – DISI, Università degli Studi di Genova
joint work with D. Ancona, M. Ancona, N. D. Matsakis

DLS'07 OOPSLA Montreal CA

October 22, 2007
Outline

1. Introduction to RPython
2. RPython idioms
3. Implementation notes and benchmarks
4. Conclusions
Dynamic languages for .NET and JVM

- .NET and JVM: widespread platforms
- Designed for static languages
- Great Python implementations: IronPython, Jython
- Much slower than e.g. C# and Java
Dynamic vs. static

Dynamic languages

- Flexibility
- Rapid development cycle
- Metaprogramming

Static languages

- Speed
- Nothing more :-)

Antonio Cuni (DSL’07 OOPSLA)
Dynamic vs. static

Dynamic languages
- Flexibility
- Rapid development cycle
- Metaprogramming

Static languages
- Speed
- Nothing more :-)

Antonio Cuni (DSL’07 OOPSLA)
RPython Quick Facts

- **Restricted subset** of Python
- Statically typed (**type inference**)
- Still allows metaprogramming
- RPython programs still run under \{C,J,Iron\}Python
- Three backends: C, .NET, JVM
- **Almost as fast** as C, C#, Java
Type inference

- Top-down, starting from an entry point; whole program analysis
- Assign the most precise type to each variable
- Fail if you try to mix incompatible types

RPython

def main():
    print add(40, 2)

def add(a, b):
    return a+b

Not RPython

def fn(flag):
    if flag:
        return 42
    else:
        return 'hello'
Type inference

- Top-down, starting from an entry point; whole program analysis
- Assign the most precise type to each variable
- Fail if you try to mix incompatible types

RPython

```python
def main():
    print add(40, 2)

def add(a, b):
    return a+b
```

Not RPython

```python
def fn(flag):
    if flag:
        return 42
    else:
        return 'hello'
```
Type inference

- Top-down, starting from an entry point; whole program analysis
- Assign the most precise type to each variable
- Fail if you try to mix incompatible types

RPython

```python
def main():
    print add(40, 2)

def add(a, b):
    return a+b
```

Not RPython

```python
def fn(flag):
    if flag:
        return 42
    else:
        return 'hello'
```
Other restrictions

- Globals are assumed to be **constant**
- Yield and generators not supported
- No special `__methods__` (except `__init__` and `__del__`)
- No run-time definition of new functions and classes
- Cannot modify classes at run-time
- Cannot change the `__class__` of an object
- **Single inheritance**, with limited support for mixins
Still pythonic, though

- No syntactic restriction
- Functions and classes are first-order values
- Exceptions work

**Lists and dictionaries**

- Work, but they must be homogeneous
- list of int, dict from string to floats, etc. are OK
- list of \emph{int and strings} is not
- Most of methods of list, dict and str are supported
Still pythonic, though

- No syntactic restriction
- Functions and classes are first-order values
- Exceptions work

Lists and dictionaries

- Work, but they must be homogeneous
- list of int, dict from string to floats, etc. are OK
- list of int and strings is not
- Most of methods of list, dict and str are supported
Init-time, translation-time, run-time

Init time

Full Python

*.py

Python interpreter

Translation time -- RPython

Live objects

Translation toolchain

Executable
RPython restrictions only apply to live objects
No restrictions about how they are created
  ➤ Full Python is allowed at init-time
Python as a metaprogramming language for RPython
Code generation considered harmful
Outline

1. Introduction to RPython
2. RPython idioms
3. Implementation notes and benchmarks
4. Conclusions
Compute complex constants

Fibonacci’s numbers

def fibo(N):
    sequence = []
a, b = 1, 1
    for i in xrange(N):
        sequence.append(a)
a, b = b, a+b
    return sequence

# computed at init-time
fibo_numbers = fibo(100)
Metaclasses run at init-time

__extend__ metaclass

class MyClass(object):
    def foo(self): ...

class __extend__(MyClass):
    def bar(self): ...

def main():
    obj = MyClass()
    obj.bar()
Dynamic classes/functions at init-time

“Static” nested scopes work

```python
def make_adder(N):
    def add(x):
        return x+N
    return add

add10 = make_adder(10)
add20 = make_adder(20)

def main():
    print add10(32)
    print add20(22)
```

Outline

1. Introduction to RPython
2. RPython idioms
3. Implementation notes and benchmarks
4. Conclusions
The Translation Toolchain

- **CPython**: *py --&gt; Python bytecode
- **FlowObjSpace**: bytecode --&gt; flow graphs
- **Annotator**: type inference on flow graphs
  - High level Python types (List(Integer))
- **RTyper**: high level types --&gt; low level types
  - lltype for C, ootype for CLI and JVM
- **Backends**: code generation
  - C, CLI (.NET), JVM
Benchmarks

- Classic Martin Richard’s test
- Available in Java, C#, RPython

<table>
<thead>
<tr>
<th>Language</th>
<th>Result</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Results on Microsoft CLR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td>6.94 ms</td>
<td>1.00x</td>
</tr>
<tr>
<td>RPython</td>
<td>7.25 ms</td>
<td>1.04x</td>
</tr>
<tr>
<td>IronPython</td>
<td>1675.00 ms</td>
<td>241.35x</td>
</tr>
<tr>
<td><strong>Results on JVM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>1.77 ms</td>
<td>1.00x</td>
</tr>
<tr>
<td>RPython</td>
<td>2.10 ms</td>
<td>1.18x</td>
</tr>
<tr>
<td>Jython</td>
<td>2918.90 ms</td>
<td>1641.80x</td>
</tr>
</tbody>
</table>

Antonio Cuni (DSL’07 OOPSLA)
Outline

1. Introduction to RPython
2. RPython idioms
3. Implementation notes and benchmarks
4. Conclusions
What’s good about RPython

- Pythonic enough to be usable
- Very fast
- Testable under CPython
Things to improve

- Originally an implementation detail
- Not designed to be user-friendly; terse error messages
- Lack of documentation/reference manual
- Lack of separate compilation
- Integration with the hosting platform
  - Good for C/Posix
  - Proof of concept for .NET
  - Doesn’t exist for JVM
About PyPy

Python in (R)Python
- High level interpreter written in RPython
- Easy to understand
- Easy to extend

Translation Toolchain
- Written in full Python
- Works as a general compiler
- Especially for interpreters (e.g. Javascript, Prolog)
About PyPy

Python in (R)Python
- High level interpreter written in RPython
- Easy to understand
- Easy to extend

Translation Toolchain
- Written in full Python
- Works as a general compiler
- Especially for interpreters (e.g. Javascript, Prolog)
Acknowledgments

- The whole PyPy Team
  - RPython is **not** mine :-)!
  - Our contributions: CLI and JVM backends
- Thanks for the attention
- Any question?