RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages *

Davide Ancona Massimo Ancona
Antonio Cuni
DISI, Univ. of Genova, Italy
{davide,ancona,cuni}@disi.unige.it

Nicholas D. Matsakis
ETH Zurich, Switzerland
nicholas.matsakis@inf.ethz.ch

Abstract

Although the C-based interpreter of Python is reasonably fast, implementations on the CLI or the JVM platforms offers some advantages in terms of robustness and interoperability. Unfortunately, because the CLI and JVM are primarily designed to execute statically typed, object-oriented languages, most dynamic language implementations cannot use the native bytecodes for common operations like method calls and exception handling; as a result, they are not able to take full advantage of the power offered by the CLI and JVM.

We describe a different approach that attempts to preserve the flexibility of Python, while still allowing for efficient execution. This is achieved by limiting the use of the more dynamic features of Python to an initial, bootstrapping phase. This phase is used to construct a final RPython (Restricted Python) program that is actually executed. RPython is a proper subset of Python, is statically typed, and does not allow dynamic modification of class or method definitions; however, it can still take advantage of Python features such as mixins and first-class methods and classes.

This paper presents an overview of RPython, including its design and its translation to both CLI and JVM bytecode. We show how the bootstrapping phase can be used to implement advanced features, like extensible classes and generative programming. We also discuss what work remains before RPython is truly ready for general use, and compare the performance of RPython with that of other approaches.

* This work has been partially supported by PyPy EC Project IST FP6-004779 and by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments.

DLS'07, October 22, 2007, Montréal, Québec, Canada.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers; D.3.2 [*Programming Languages*]: Language Classifications—object-oriented languages

General Terms Languages, Performance

1. Introduction

Java and .NET are two widespread and successful platforms: they are both based on two high performance, very reliable virtual machines, respectively the JVM and the CLI, which allow the development of portable and interoperable applications. Both virtual machines, however, have been designed with statically typed languages in mind, and their native object model force objects to have a fixed set of fields and methods. While these static guarantees enable earlier error detection and efficiency, they also come at the cost of a loss of expressive power.

On the other hand, information technology is quickly moving to scenarios where computing needs to be ubiquitous, pervasive, and highly dynamic. In order to thrive in this environment, modern software systems must be dynamically adaptable and updatable. This fact explains the growing interest in dynamically typed languages, like Python, JavaScript and Ruby. Both the JVM and the CLI have also felt the pressure to add dynamic features, such as support for dynamic typing [28].

There have been several implementations of dynamically typed languages for both the JVM and the CLI, such as Jython [15], IronPython [13], JRuby [14], and Rhino [24]. While these implementations are very usable, and generally feature well-developed integration with their hosting environments, they are significantly slower than Java or C# (see Section 4.4). This is generally because the primitives provided by the JVM and the CLI are not sufficiently dynamic to be directly usable, and so the implementors are forced to emulate the dynamic object model in an interpreter-like fashion.

Restricted Python (subsequently called *RPython*) is an attempt to, in the words of Mejier and Dryton, "seek the

golden middle way between dynamically and statically typed languages" [22]. It is a proper subset of Python, restricted in a way that enables easy analysis and efficient code generation, but still maintains many of Python's hallmark features. The result is a language that is more expressive than C# and Java, but which does not compromise runtime efficiency. RPython was initially designed for the specific purpose of implementing PyPy [25] (a Python interpreter written in Python), but it has grown into a full-fledged language in its own right.

Currently, RPython can be used in many contexts: to develop stand-alone programs, such as the Standard Interpreter itself; to write highly efficient extension modules for CPython, which could only be written in C in the past; to develop dynamic web applications without the need to write JavaScript code; to produce efficient libraries of classes and functions to be used by other .NET and Java programs. In particular, RPython can be the ideal companion for all those CPython, IronPython and Jython developers that so far have been forced to write the parts of their programs that need high performance in C, C# or Java.

The main restrictions that must be obeyed for a Python program to be considered RPython are:

- Dynamic features like class modification can be used in full, but only during an initial, bootstrapping phase. This phase is used to construct the final RPython program that is actually executed.
- The final RPython program to be executed must be well-typed, according to the type inference rules of RPython. Note that every type correct RPython program is also a valid Python program.
- Only single inheritance is permitted, though there is support for mixins [10], which offer many of the advantages of multiple inheritance.

The contributions of this paper are two-fold. First, we present an overview of RPython and give several examples which demonstrate the many meta-programming techniques made possible by its preprocessing step.

Second we describe the design and implementation of the JVM and CLI back-ends we have developed, and show how the design of RPython allows a smooth translation of the language for these platforms.

The paper is structured as follows. Section 2 gives an overview of RPython where several examples demonstrate its expressiveness, despite the limitations imposed by its static type system. The next two sections describe the architecture of the RPython compiler; Section 3 outlines the design of the front-end, whereas Section 4 describes some of the details of our implementation of the JVM and CLI back-ends. Section 5 discusses some issues both at the design and implementation level that should be addressed in the future in order to make RPython a more usable and useful language. Section 6 gives a brief overview of the related work. Finally, Section 7 draws some conclusions.

2. An overview of RPython

2.1 The *PyPy* project

The *PyPy* project aims to create a platform that makes it easy to experiment with different virtual machine designs, but without sacrificing efficiency. This is achieved by separating the semantics of the language being implemented, such as Python or JavaScript, from low-level aspects of its implementation, such as memory management or the threading model. A complete interpreter is constructed at build time by weaving together the interpreter definition and each low-level aspect into a complete — and efficient — whole [26].

In order to best separate the low-level aspects of an interpreter from the language semantics, the interpreter definition itself must be written in a high-level language that abstracts these details away, and, at the same time, can be statically analyzed and compiled easily. This language is RPython.

Thus, PyPy is composed of two major parts: the *Standard Interpreter* and the *Translation Toolchain*. The Standard Interpreter is a full-featured Python interpreter written in RPython. The Translation Toolchain analyzes the interpreter, performs type inference, and produces an efficient executable version.

It is important to emphasize that the Translation Toolchain is a general compiler that can be used for other projects beyond the Standard Interpreter, although it has been heavily optimized to work well with the latter. In fact, interpreters for such languages as Prolog and JavaScript also exist in various states of completeness.

2.2 RPython as a language

As explained above, RPython was born as an implementation detail of PyPy. During the development of PyPy, RPython proved to be a very convenient language for development, and eventually grew into a useful end-product in its own right.

Because RPython is a subset of Python, every RPython program can run unmodified on the top of the Python interpreter. This allows programmers to take advantage of all the introspective features of Python during testing and debugging.

Python is a dynamically typed language: type information is attached to the objects, rather than to the method parameters, local variables, or return values. For instance, the following is a valid Python function which, depending on the value of its `x` parameter, returns either an integer or a string:

```
def foo(x):
  if x: return 42
  else: return 'bar'
```

Since this flexibility comes at the cost of a loss of efficiency, the above definition is not correct in RPython; indeed, all those dynamic features that cannot be efficiently

implemented have been removed. Therefore, the static type system of RPython forbids using values of incompatible types together; however, this is not a serious limitation as experience has shown that well-written Python programs can be conformed with the RPython type system with few changes.

A more significant limitation of RPython is that it is not possible to dynamically change class definitions, by adding or removing methods and fields. In RPython, as in Java or C#, each object belongs to a class and each class has a fixed set of fields, methods, and superclasses (see Section 2.3). Although this limitation significantly reduces the expressive power of RPython when compared to full Python, it is necessary for generating efficient executables; furthermore, special care has been taken to make RPython "as pythonic as possible", meaning that developers can still use most of the typical patterns and idioms they use when developing in Python.

Despite the above restrictions, RPython is still much more dynamic and expressive than most of the static mainstream languages such as Java and C#. It supports all the features one could expect to find in any modern object-oriented language such as classes, single inheritance and exceptions, and it is statically typed; furthermore, it also supports a lot of features that are ordinary for Python programmers but that are usually not found in those languages: limited support for mixins, first-order function and class values, limited use of bound methods, metaclasses.

2.3 Type system and object model

RPython supports the following primitive types, with some variants: `SomeInteger` (signed, unsigned, non-negative), `SomeFloat`, `SomeBool`, `SomeChar`, `SomeString`[1]. While in Python each of these types is an object, in RPython these types can only be used in restricted ways. This ensures that it will be possible to represent instances of these types using their native counterparts during execution. For example, integers can be stored as a scalar `int` rather than some kind of wrapped integer object.

Although Python does not distinguish between strings and characters, RPython uses different types for them; during type inference, strings whose length is exactly one character are annotated with the type `SomeChar`. This allows backends to use the native types for chars, when available (see 3.2.1).

In addition to the primitive types, there are few built-in container types: `SomeTuple`, `SomeList`, and `SomeDict`. These types are generic types, that is, they are parameterized by the types of the items which they store.

`SomeTuple` represent *tuples*, which are used in both Python and RPython to group together small sets of non-homogeneous objects. Tuples are found in many languages, and can be thought of as a read-only, anonymous record. Their items are accessed either by index or by **tuple unpacking**. They are commonly used to return multiple values from a function, as in this example:

```
def divmod(a, b):
    return a/b, a%b
quot, rem = divmod(10, 3) # unpacking
```

`SomeList` objects are used to store mutable sequences of items, all of which must be of the same type. They take the place of both fixed-size arrays and growable sequences such as the `ArrayList` class from Java and .NET. The compiler determines whether the size of a list is fixed or variable, so that the back-end can select the most efficient representation. `SomeList` objects provides most of the Python methods for lists, such as `sort`, `reverse`, `append` and `pop`.

Finally, `SomeDict` is the RPython equivalent of a Python dictionary; dictionaries represent a mapping between keys and values and are usually implemented with a hashtable. As with lists, RPython dictionaries need to be homogeneous, i.e. all the keys must belongs to the same type, as well as all the values (the type of the keys does not necessarily need to be the same as the type of values).

In addition to basic types, developers can define their own classes. As happens in Python, the fields of an RPython class are not declared as they would be in a traditional language. Instead, the fields (and their types) are inferred automatically based on the program. For example, consider the following code:

```
class MyClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

def foo(x):
    obj = MyClass("Hello world", x)
```

When analyzing the function `foo`, the compiler can detect that instances of `MyClass` contain a string field named `a` and an integer one named `b`, given that `x` was previously annotated as an integer.

RPython only supports single inheritance; however, it also supports *mixin* definitions, which allow common methods to be shared among many classes, without affecting the inheritance hierarchy. Intuitively, mixins can be seen as a collection of methods that are cut-and-pasted into the class definition. The syntax for declaring and using a mixin is similar to multiple inheritance, except that mixin classes are marked by a special `_mixin_` attribute.

The following example demonstrates the composition of two mix-ins, `Displayable` and `Addable`:

```
class Displayable:
    _mixin_ = True
    def display(self):
        print 'value = ',self.value

class Addable:
```

[1] The prefix `Some` reflects the naming convention used internally by the annotator.

```
    _mixin_ = True
    def add(self, x):
        return self.value + x

class Number(Displayable,Addable):
    def __init__(self, value):
        self.value = value

class String(Displayable,Addable):
    def __init__(self, value):
        self.value = value

def main():
    n = Number(40)
    s = String('Hello ')
    print n.add(2)         # 42
    print s.add('world!') # Hello world
    n.display()            # value = 40
    s.display()            # value = Hello
```

RPython mixins inherit their semantics from their Python equivalents [10] and implement a mixture of the semantics of traits [8] and mixins as described by Bracha and Cook [3]. Like traits, RPython mixins do not affect the inheritance hierarchy and the methods of the class take the precedence on the methods of the traits. Like standard mixins, the order in which RPython mixins are composed is relevant.

Finally, classes and methods are first-order entities, i.e. they can be stored and passed around to be called/instantiated later, as shown in figure 1.

```
def add(x, y): return x + y
def sub(x, y): return x - y
def proc(f, x, y): return f(x, y)
```

Figure 1. A simple RPython program using first-class functions

2.4 Initialization-time, translation-time and run-time

In Python, functions and classes are not defined by declarations, but by executing the `def` and `class` statements, which have the side effect of creating a function or class object, respectively. When a module is imported into the interpreter, its top-level statements are executed and the resulting objects are collected into the namespace of the module.

Unlike most compilers, the translation of RPython programs does not start from the source files, but rather from live Python objects that have been created and initialized by the standard Python interpreter. Thus, the life cycle of an RPython program is divided into three phases, as shown by Figure 2.

Initialization: the process of constructing and initializing Python classes, functions and constants to be compiled.

Translation: the process of analyzing the program as a whole, inferring types and producing an executable.

Run: the execution of the output produced by the *translation* step.

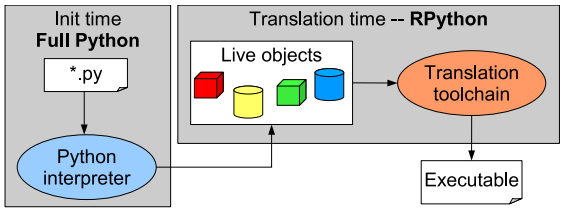

Figure 2. From Python sources to compiled executables.

Because the *translation step* only examines the objects once they have been fully created, it is not aware of how they were generated. This means that at initialization-time we can exploit the full power of Python, without restrictions, to dynamically build these live objects; this includes, but it is not limited to, `exec`, nested scopes, and metaclasses.

In other words, we could say that RPython programs are not *written* in the form of source code, but they are *generated* by the Python statements that create those live objects; thus, we can think of Python as the **meta-programming** language for RPython. This allows for a wide range of language extensions.

2.5 Useful RPython programming patterns

This section contains an overview of useful techniques that take advantage of the Python pre-processing step to make programs simpler and easier to maintain. Some of what we describe in this section could be implemented in Java or C# today using reflection: however, this entails a heavy runtime price, whereas in PyPy these patterns operate purely in a pre-processing phase, with no time penalty at runtime.

2.5.1 Initialization of complex constants

RPython's meta-programming capabilities can also be very helpful for computing complex constants at initialization time.

For example, suppose that your program needs to frequently use the first `N` *Fibonacci's numbers*; the usual solution in C# or Java is either to read those numbers from a file where they were previously stored, or to pre-compute them as soon as the program starts, usually inside the *static constructor* of some class. This solution might be problematic if this computation takes a long time, because it would impact on the start-up time of the program.

With RPython, you can simply do the computation during the initialization phase, and store the results into a constant:

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

# pre-compute the first 100 numbers
fibo_numbers = fibo(100)
```

Note that, even if we called `fibo` at init-time, nothing prevent us from calling it at run-time as well. *The same function serves equally well for both meta-programming and programming*.

2.5.2 Extending the language through metaclasses

Python, like Smalltalk [17], CLOS [16], and Objective-C [6], includes extensive reflective capabilities. Reflection is the activity performed by a program when doing computations about itself [20]. This includes both introspection (state and structure observation) and intercession (the alteration of structure and behavior). In a reflective system, objects can be represented by other objects, usually referred to as meta-objects (for example the class of a class object is a meta-class).

In Python, custom metaclasses can be used to change the meaning of a class declaration, and therefore affect what happens when new instances are created, and so on. This customization takes place at class-definition time, i.e. when the **`class`** statement is executed. Because RPython does not allow classes to be defined after the initialization period, metaclasses run entirely in the Python interpreter and are automatically fully supported.

Explaining the details of metaclasses in Python is beyond the scope of this article; for more information, see [31]. To showcase the almost infinite possibilities, we present an example using the `extendabletype` metaclass. This metaclass is widely used in PyPy to allow programmers to *extend* an already defined class with new methods.

Suppose that we have a class hierarchy of business objects. Each object can do two things: save itself into a database, and present itself in a GUI. The resulting class structure might look like in the following code fragment:

```
class Root:
  def save(self, db):      ... # abstract
  def show(self, window): ... # abstract
class MyFirstObject(Root):  ...
class MySecondObject(Root): ...
```

This design is not optimal, because it ties together three unrelated aspects: the business model, the storage and the presentation. Several approaches have been proposed to solve this problem, such as the *visitor pattern* [11], but most of them are, in fact, ways to workaround the inability to define classes incrementally.

The `extendabletype` metaclass solve this problem by letting the programmer to split the same class definition into different files, so that only related aspects are grouped together:

```
# file model.py
class Root:
    __metaclass__ = extendabletype
    ...
class MyFirstObject(Root):  ...
class MySecondObject(Root): ...

# file db.py
class __extend__(Root):
  def save(self, db):  ... # abstract
class __extend__(MyFirstObject):  ...
class __extend__(MySecondObject): ...

# file gui.py
class __extend__(Root):
  def show(self, window): ... # abstract
class __extend__(MyFirstObject):  ...
class __extend__(MySecondObject): ...
```

The `extendabletype` metaclass allows most class definitions to complete normally, unless the class being defined has the name `__extend__`. In that case, instead of creating a new class object, the metaclass adds the newly declared methods and fields to the already-existing class being extended (in this example, either `Root`, `MyFirstObject`, or `MySecondObject`). The code makes heavy use of the introspective and dynamic features of Python, but since it runs during the initialization phase it is still usable in RPython.

C# 2.0's *partial classes* offer the same benefits, as well as *MultiJava* [5]; the key point of this example is that in RPython this behavior is simply *metaprogrammable*, while for C# and Java the only way to do it is to extend the language or to run a preprocessor.

2.6 A complete example

This sections shows how the features of RPython help to write a real (though small) program: a simple parser and interpreter for reverse polish notation. The program first parses the command line arguments to create an expression tree, and then evaluates the tree into an integer result and prints it. For sake of simplicity, we deliberately omit the code handling syntax errors. Although the example only includes addition and subtraction, it can be trivially extended to deal with more operators.

The example shown in Figure 3 uses a variety of meta-programming techniques to generate the code for handling the various operators. Of course, since the example is so small, it would be easier just to inline the code that is required. However, this approach makes it trivial to add new operators, or to separate the operator definitions from the parser, and thus is important as the language to be parsed becomes more complicated.

The core of the program is the `parse` function. It takes a list of tokens as input and builds an abstract syntax tree representing the expression. To locate the appropriate class it relies on a precomputed dictionary, `OPCODES`, which maps each operator token (`'+'` or `'-'`) to its corresponding class. This exploits the fact that RPython classes are first-order values (see Section 2.3).

The program relies on an initialization step performed by the function `build_opcodes`. This function does two things: first, it adds an automatically-generated `eval` method to each binary operator class, and then adds the class to the `OPCODES` dictionary. Both steps rely on the convention that the *doc-*

```
# AST classes
class Expr:
  def eval(self):
    raise NotImplementedError

class Number(Expr):
  def __init__(self,n): self.n = n
  def eval(self):       return self.n

class BinaryExpr(Expr):
  def __init__(self,l,r):
    self.l = l
    self.r = r

# operators: the docstrings contain the
# symbol associated with each operator
class Op_Add(BinaryExpr):
  '+'
class Op_Sub(BinaryExpr):
  '-'

def main(argv):          # entry-point
  e = parse(argv[1:])
  print e.eval()
  return 0

def parse(lst):
  stack = []
  for token in lst:
    try:
      node = Number(int(token))
    except ValueError:
      op_cls = OPCODES.get(token, None)
      y, x = stack.pop(), stack.pop()
      node = op_cls(x, y) # instantiation
    stack.append(node)
  return stack[0]

# INIT-TIME only: build the table of
# opcodes and add the 'eval' methods
def gen_eval(ch):
  code = """
  def eval(self):
    return self.l.eval() %s self.r.eval()
  """
  d = {}
  exec code.strip() % (ch) in d
  return d['eval']

OPCODES = {}
def build_opcodes():
  for name, value in globals().items():
    if name.startswith('Op_'):
      value.eval = gen_eval(value.__doc__)
      OPCODES[value.__doc__] = value
build_opcodes()
```

Figure 3. A simple RPN calculator

string[2] of each operator class contains its corresponding symbol.

[2] *docstrings* are the Python way to attach the documentation to functions and classes. Compared to other approaches such as *JavaDoc*, the main difference is that they are also available at run-time, and they can be accessed via the __doc__ attribute.

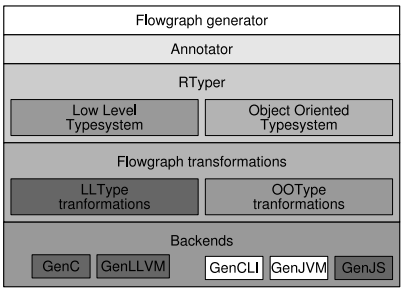

Figure 4. The compiler architecture.

The `eval` method is generated via a call to the helper routine `gen_eval`, which creates a string of Python code that performs the desired computation and uses **`exec`** to compile this string into a Python method.

Adding the class object to the `OPCODES` dictionary is done by simple assignment, using the class docstring as the key and the class object itself as the value.

3. RPython Front-end

The overall architecture of the compiler is summarized in Figure 4. The RPython source objects are first analyzed by the *Flow Graph Generator*, which creates flow graphs representing the control flow, but without any type information. These flow graphs are analyzed by the *Annotator*, which adds annotations about the type and other details of the values in the program. In compiler terminology, these annotated flow graphs correspond to a High-Level IR (intermediate representation), as they are defined in terms of the source language. These annotated flow graphs are then lowered by the RTyper to lower-level graphs more suitable for code generation. These lower-level graphs correspond to a mid- or low-level IR in a typical compiler, as they target a simplified model of the actual machine for which code is eventually generated. Finally, the chosen back-end is used to produce actual executable code based on these lower-level graphs. In Section 4, we discuss two such back-ends, for the .NET and JVM platforms, in detail.

The rest of this section provides an overview of the Flow Graph Generator and Annotator; for full details, however, the reader is referred to the corresponding publications of the PyPy project [26, 25].

3.1 Flow Graph Generator and Annotator

The *Flow Graph Generator* uses abstract interpretation to transform the live Python objects which form the source code of an RPython program into a set of control flow graphs. The resulting flow graphs are in an untyped Static Single Information (SSI) form, and the operations are at a very high level, corresponding directly to their RPython origins. SSI [1] is an intermediate format derived from SSA

which in addition to traditional variable definition points, provides new names for each variable at each point where the analysis may obtain information about the value in the variable; in this way it is possible to propagate information directly from information definition sites to information use sites.

The Annotator exploits the SSI format to perform a static analysis of the flow graphs which assigns each variable a specific RPython type (see also Section 2.3) which describes all of the possible values the variable may have at run-time. If no suitable RPython type can be found for a variable, then the program is rejected and processing stops.

Note that both the Flow Graph Generator and the Annotator perform a global analysis of a program as a whole. The user is required to supply a main function that serves as an entry point, and a corresponding set of type annotations for their parameters. While this approach works fine for PyPy's main purpose of writing interpreters, it has drawbacks when attempting to use RPython as a more general purpose language (see Section 5).

The types inferred by the annotator correspond directly to the RPython types. For example, a variable may be annotated with the type `SomeList(SomeInteger())`; such a list exposes most of the operations of a standard Python list, with the difference that the types of its elements are known to be integers, rather than being any object, as in Python.

3.2 RTyper

RTyper is an abbreviation for "RPython low-level typer" and is the component which bridges between the high-level flow graphs of the front-end and the low-level details required by the back-end.

While it should be possible in theory to write a back-end which operates directly on the high-level flow graphs, it is generally too complicated to be feasible without an intermediate processing stage. In addition, the RTyper allows sharing of a considerable amount of code between back-ends for similar platforms, as happens for GenCLI and GenJVM.

The RTyper performs two main kinds of transformations on the typed flow graphs. First, each annotation is translated from a high-level RPython type into an appropriate lower-level type. Second, high-level operations contained in a block are broken down into one or more lower-level operations.

In fact, the RTyper itself is customized by a specific type system, which determines the exact lower-level types and operations to be used. This allows the back-ends for lower-level targets, such as the C language, to obtain a lower-level representation than the back-ends for higher-level targets, such as the JVM or CLI. The next section discussed these type systems in more detail.

3.2.1 The *lltype* and *ootype* type systems

So far, two type systems have been developed for use with the RTyper: *lltype*, which is intended for lower-level targets and built around struct, array, and pointer types, and *ootype*, which is intended for high-lever, virtual-machine targets, and which uses class and object types.

To clarify the difference between the two type systems, consider again the annotation `SomeList(SomeInteger())`. In *lltype* this RPython type would be mapped into the following struct type:

```
struct list {
    int length;
    int* items;
};
```

By contrast, the *ootype* will translate that annotation into `List(Signed)`, where `Signed` is the low-level type for `SomeInteger()`. The high-level type `SomeList` differs from the low-level type `List` because the former supports operations typical of RPython,[3] while the latter is designed to have only operations that are presumed to be directly available in the target platform. Typically, instances of type `List` would be mapped to a built-in library class for representing lists, such as the `ArrayList` class offered by Java and C#.

The full description of the *ootype* type system is available in [7]; it supports a set of *primitive types* for numbers and strings (e.g., `Signed`, `Unsigned`, `String`), some *collection types* (`List`, `Dict`), first-order functions and classes (`Class`, `StaticMethod`) and of course user-defined classes.

The object model of *ootype* is Java-like, meaning that classes have a static set of typed methods and fields, only single inheritance is supported and the class hierarchy is rooted by the predefined class `ROOT`; moreover, every method is implicitly *virtual*, that is, can be overridden in some subclasses; furthermore, methods can be *abstract*.

Classes, attributes and methods do not have access restrictions: *ootype* is only used internally by the translator, so there is no need to enforce accessibility rules.

Moreover, as for their RPython counterparts, collection types such as `List` and `Dict` are *generic* in order to allow back-ends such as GenCLI to exploit native support for generic programming; back-ends for platforms like Java need to implement a type erasure technique similar to that performed by the Java compiler.

4. Back-end Implementations

4.1 Low-level and object-oriented back-ends

As mentioned in Section 3.2.1, the RTyper can specialize the high-level graphs using either the *lltype* or *ootype* type system. Like the RTyper type systems, the back-ends are also divided into two categories, low-level and object-oriented.

Currently, there are two low-level back-ends: GenC, which produces C source code for the *gcc* compiler, and GenLLVM, which generates LLVM bytecode [18]; and three

[3] As an example of RPython-specific semantics, consider the extended index notation for accessing list elements: e.g. `mylist[-1]` refers to the last element of the list, and `mylist[2:5]` extracts a *sublist* containing the second, third and fourth element.

object-oriented back-ends: GenCLI, which targets the CLI virtual machine, GenJVM, for the Java virtual machine, and GenJS, which produces JavaScript code to be run into a web browser.

In the following sections we will discuss only GenCLI (Section 4.2) and GenJVM (Section 4.3). The details of the other back-ends have been already described in a previous paper [26] and in the PyPy technical reports [25, 7].

The Java Virtual Machine (JVM) [19] and the CLI [9] share many similarities, which enables GenCLI and GenJVM to share a substantial amount of code.

4.1.1 JVM and CLI in a nutshell

The Java and CLI virtual machines look very similar for the purpose of writing a back-end for RPython. They are stack-based machines, both providing a rich, statically typed, object oriented type system.

Both machines require that all code be encapsulated within a class.[4] Both support single inheritance between classes, but allow multiple inheritance in the form of *interfaces*, which are similar to an abstract base class in C++ (i.e., a class where all methods are abstract, and which contains no fields).

Typically, code loaded into the JVM and CLI must first undergo a verification step before it can be executed. This step assigns a static type to each value in the program. Method calls must be annotated with the type of the receiver, of each argument and of the returned value to perform type-checking at runtime. This static typing requirement is precisely what makes it difficult to port a dynamically-typed language like Python to these machines.

Of course, there are some differences between the JVM and the CLI as well. Two important differences that affect RPython's code generation are:

- In order to be thrown as an exception, the JVM requires that an object have a type which descends from `java.lang.Throwable`, while CLI allows any object to be thrown as an exception.
- The CLI supports *generic types*, meaning that a single class definition can be parameterized by another types. For example, an object of type `List` might be parameterized by the type of the objects which it contains. When a `List` object is created, the type of the objects which will be stored in it is also specified; the CLI then clones the generic type definition as needed to match these type parameters. While the JVM does not support generics directly, they can be emulated using the technique of *erasure*[4] (just as the Java language itself does).

For further details about the specifications of the two virtual machines, the reader is referred to [19] and [9].

[4] This is not strictly true for CLI, since it allows *global static methods*, not enclosed by a class; this feature is not portable across different languages, though.

4.1.2 Translation from RPython to CLI and JVM

Unlike full Python, most constructs in RPython have a direct correspondence with a similar construct in the CLI and JVM. For example, RPython classes are always mapped to a single CLI/JVM class,[5] and RPython function calls are implemented with the native CLI or JVM instructions, `call instance` and `invokevirtual`, respectively.

RPython containers such as `SomeList` and `SomeDict` are mapped directly to their equivalent classes from the platforms standard library (`List<T>` and `Dict<K,V>` for the CLI and `Vector` and `HashTable` for the JVM). In the case of the CLI, we take advantage of the CLI support for generic classes; for the JVM, we must use casts and boxing when adding and removing items from containers.

However, there are other characteristics of RPython that are not mapped so directly into the CLI or the JVM and require more careful treatment, as explained in sections 4.2 and 4.3.

4.1.3 Register- to stack-machine translation

As shown in Section 3.1, RPython programs are represented as flow graphs expressed in SSI form, a register-based intermediate representation.

Since both the CLI and the JVM are stack-based virtual machines, we are forced to convert between the two. Thus, we developed a common piece of code that can group the register-based instructions into trees to allow for reasonably efficient stack-based code; the details of these techniques are not the topic of this paper.

Currently, this algorithm is fully integrated into GenCLI but not yet into GenJVM; efforts are underway to let GenJVM to exploit the transformation form SSI to tree-based form.

4.2 GenCLI: The .NET back-end

GenCLI is the back-end that targets the .NET virtual machine. It tries to be fully compatible with the two most widespread implementations, i.e. the Microsoft Common Language Runtime (CLR) [23] and Mono [29].[6]

4.2.1 First-order functions and classes

A significant difference between *oootype* and CLI is that the latter does not support first-order functions. .NET *delegates*[7] are exploited to encode them: a new delegate type is created for each different signature of the functions used in the program, then such functions are wrapped inside *delegate objects* and passed around: finally, to call such a wrapped function we simply call the *Invoke* method of the delegate.

[5] With one exception: see Section 4.3.2.

[6] Mono is an open source .NET implementation for Windows and Linux

[7] *Delegates* are the safe .NET equivalent to function pointers; they wrap a static or instance method call inside an object that can be passed around and invoked later (for example, callback functions).

Finally, *ootype* also supports classes as first-order objects, while CLI does not: to solve the problem we can pass around the instance of `System.Type` which corresponds to the class itself. Then we can dynamically create an instance of that class by calling the helper method shown below, which is written in C# and uses reflection to get the default constructor and call it,

```
public static object RuntimeNew(Type t) {
  return t.GetConstructor(new Type[0])\
                .Invoke(new object[0]); }
```

4.2.2 Accessing the .NET libraries

One of the major advantages of using a .NET compiler is the access to a vast library of existing classes.

GenCLI allow the programmer to access the external .NET libraries, although the interaction between the two worlds is still not perfect.

In particular, the current syntax is less than ideal, since GenCLI does not yet support *indexers* or *properties*[8], and the only way to call them is to directly call the underlying methods; for example, to access the elements of instances of `ArrayList`, you must explicitly call the `get_Item` and `set_Item` methods, instead of using the square bracket syntax.

Although in Python and RPython methods can not be overloaded,[9] GenCLI allows the programmer to call overloaded methods defined in external libraries, using the types of the arguments at the call-point to determine the best match, as for instance C# does.

4.3 GenJVM: The JVM back-end

Currently the JVM back-end has not been completed yet, and cannot translate all valid RPython programs; however, most conceptual hurdles have been overcome, and what remains is the implementation of all corner cases. After that, another important piece of remaining work will be to generate more optimized bytecode for better performance.

4.3.1 First-order functions and classes

Like the CLI, the JVM does not directly support the kind of first-class functions and classes which RPython offers. The JVM back-end emulates these features by creating objects encapsulating each stand-alone function, and using reflection to emulate first-class classes. The JVM solution is very similar to the CLI solution, except that it replaces the use of delegates with an abstract base class, as delegates are not available on the JVM.

Figure 1 depicts an RPython program which uses first-class functions to choose between the functions `add` and `sub`. Figure 5 contains Java code similar to that which our backend would generate.

As can be seen from the classes `AddFunc` and `SubFunc` in Figure 5, the back-end wraps each function that is stored in a variable in a class which can be instantiated. In addition, one abstract base class is created for each unique set of argument and return types; in this example, that is the class `IntInt`, and, as shown in the class `ProcFunc`, it is used as the static type for pointers to functions that take two arguments. The choice between an abstract base class and an interface is arbitrary; we chose abstract base classes for performance reasons.

```
abstract class IntInt {
    abstract int invoke(int x, int y);
}
class AddFunc extends IntInt {
    int invoke(int x, int y)
    { return x+y; }
}
class SubFunc extends IntInt {
    int invoke(int x, int y)
    { return x-y; }
}
class ProcFunc {
    int invoke(IntInt f, int x, int y)
    { return f.invoke(x, y); }
}
```

Figure 5. Java code demonstrating how first-class functions are translated

First-order classes are much easier to implement than functions. This is because the only operation which RPython permits on a class object is to instantiate it. In the JVM, as in the CLI, this can be easily implemented using reflection. Pointers to RPython classes are therefore translated into instances of `java.lang.Class`, which can be used to create new instances of a class without specifying the class name statically.

4.3.2 Exceptions

In Java, all classes are subclasses of `java.lang.Object`;[10] one particular subclass, `java.lang.Throwable`, is special, because it is the ancestor of all exception types. Attempts to throw any object as an exception whose class does not descend from `java.lang.Throwable` are rejected by the verifier. RPython has a similar class structure. Like Java, it has a class, `Object`, at the root of the type hierarchy, and a distringuished subclass, `exceptions.Exception`, from which all exceptions are derived. While this parallel structure is necessary to allow RPython exceptions to be thrown by the JVM, it is not sufficient by itself.

The problem arises because each RPython class, including `Object`, is translated into a new Java class. The translated versions of each RPython class, therefore, form a sub-tree

[8] Properties and indexers are a form of syntactic sugar supported by C#, that allow methods to be invoked to simulate field access and array dereferencing.

[9] *Method overloading* allows the programmer to use the same name for different methods, as long as their signature are different.

[10] To distinguish between Java's `java.lang.Object` and RPython's `Object` class, we always use the fully-qualified `java.lang.Object` to refer to the Java class

within the Java class hierarchy. If this translation were not carefully handled, the result would be that RPython exceptions could not be thrown using the JVM exception handling mechanisms, because all RPython classes, including exceptions, would be a sibling of the `java.lang.Throwable` sub-tree.

To avoid this problem, the JVM back-end treats the RPython class `Object` specially. Unlike other RPython classes, `Object` is actually translated into a Java *interface*, which has two implementations. The first implementation descends from `java.lang.Object`, and is used as the superclass for all RPython classes *except* for exceptions.Exception. The second implementation descends from `java.lang.Throwable`, and is used as the superclass for `exceptions.Exception`. This effectively splits the RPython hierarchy into two sub-trees. The fact that both implementations of `Object` implement the same interface means that pointers which may point to any RPython object can use this interface as their static type and retain type safety.

4.4 Benchmarks and Comparisons with Other Compilers

Comparing the performances of a new language against others is always a difficult task. First, it is difficult to choose the right competitors to compare against, especially when the new language does not fit neatly into any existing category; moreover, the choice of the benchmarks can dramatically affect the results.

We think that RPython lies in a middle region between completely static languages, like C# and Java, and completely dynamic languages such as IronPython and Jython. Thus, in this section we will show how RPython compares to those languages. It is important to emphasize that the numbers serve only to validate that the restrictions placed upon RPython enable us to generate efficient code, not as a general measure of the performance of each language. In particular, the concrete implementation of the virtual machine can dramatically affect the overall performance of the language.

As a benchmark, we used the classical Martin Richards's benchmark [21], which was originally written for BCPL and then ported to a number of languages, including RPython, Java and C#. In particular, the RPython version can either be translated to CLI and JVM or run directly on top on IronPython and Jython. The benchmark exercises particularly object oriented features such as object instantiation and method calls; since we used only one benchmark the results are only indicative, but we think they give a good representation of overall RPython performances. In the future, we will add some micro-benchmarks to spot eventual bottlenecks.

The following table summarizes the results; the values are expressed in milliseconds and represent the "average time per iteration" (the smaller the better). For CLI, we ran the benchmarks both against Mono on Linux and Microsoft CLR on Windows XP. Linux tests have been run on a machine with a Intel Core Duo 2.5 GHz CPU and 2 GB of RAM; Windows tests have been run on the same machine under VMWare. For JVM, we ran the tests only under Linux, on the same machine.

Language	Result	Factor
Results on Microsoft CLR		
C#	6.94 ms	1.00x
RPython	7.25 ms	1.04x
IronPython	1675.00 ms	241.35x
Results on Mono		
C#	4.19 ms	1.00x
RPython	9.63 ms	2.30x
IronPython	1509.41 ms	360.24x
Results on JVM		
Java	1.77 ms	1.00x
RPython	2.10 ms	1.18x
Jython	2918.90 ms	1641.80x

Table 1. Results on Microsoft CLR, Mono, and JVM

It is interesting to compare the results on Microsoft CLR and on Mono in Table 1: although the Microsoft CLR executes both the C# and RPython version of the benchmark at almost the same speed, on Mono the RPython version is more than 2 times slower. We think this is because Mono JIT is not able to effectively optimize the bytecode produced by GenCLI, which differs quite a bit from what the C# compiler generates.

Regardless the CLR implementation, RPython is much faster than IronPython, proving that RPython restrictions really lead to a big improvement in performance.

The performance difference between the various languages on the JVM is more dramatic than on the CLI. RPython for JVM is slightly slower, relative to Java, than its CLI counterpart is, relative to C#. We believe this is because the JVM back-end is newer, and does not exploit all possible optimizations (see for example Section 4.1.3), and we are confident that we can improve it until we get the same results as RPython for CLI.

5. Towards a Usable Language

As mentioned earlier, RPython was initially designed for the specific purpose of implementing PyPy, a Python interpreter written in Python. Although it has grown into a more general purpose language, the implementation of RPython still needs a number of improvements. This section discusses the design and implementation issues that must be addressed to make RPython usable in a wider variety of contexts.

In the short term, improvements are needed to make the front-end more user-friendly, complete the JVM back-end, and further optimize the generated code.

In the longer term, there are more challenging issues to be addressed. Perhaps the most important are implementing separate compilation, improving RPython interoperability with other languages, and improving the type system to

increase RPython expressiveness without compromising its efficiency.

5.1 Separate compilation

Currently, the RPython toolchain relies on a full-program analysis, starting from a known entry point, to create the annotated flow graphs that represent the program. As a result, type-checking is not compositional. A function or class declaration which is accepted in one context may be invalid in another, depending on how it is used. This makes separate compilation rather difficult.

There are at least two approaches to solve this problem. The first is to require the programmer to annotate the types on all "public API" functions. This is relatively simple, but is contrary to the "Python philosophy". Another more difficult solution would be to implement a compositional type inference algorithm, such as the one used for JavaScript [2]. Of course, this would require a major revision to the Annotator.

5.2 Interoperability

The fact that RPython can be compiled for use on the CLI or JVM naturally invites the question of how interoperable it is with programs written in other languages running on the same virtual machine.

It is relatively easy to allow RPython code to access classes written in C#/Java. The CLI back-end already supports this, and support for the JVM back-end is planned for the near future.

On the other hand, allowing C#/Java code to access RPython classes presents a larger challenge and bears more investigation. We do not currently know how best to approach this problem; however, it is similar to the problems faced by any attempt to allow statically-typed languages to interface with their dynamic or type-inferred counterparts.

5.3 Increasing the expressive power

The current RPython type system is fairly standard and simple. One interesting area for future improvement would be to enhance the type system to increase the flexibility of the language, and allow it to typecheck a wider subset of Python programs than it does currently.

One of the main limitations of the current type system of RPython is that it does not directly support generic methods or functions. Let us consider, for instance, the following code:

```
def id(x): return x
print id("one"),id(1)
```

Clearly, the argument to `id()` cannot be assigned a single type, therefore type inference succeeds only if the user explicitly annotates the function as polymorphic. This causes the function to be cloned for every incompatible argument type. In the example above, a separate copy of `id` would be made to handle integers and strings. If the type system were extended to support generic functions, however, a single copy of the function would often suffice. Even better, such an extension to the type system would not compromise the efficiency of other programs that do not require it.

On the other hand, other kinds of type system extensions could imply a negative impact on the performance of RPython programs. For instance, it would be possible to define a more sophisticated type inference algorithm that allows the dynamic addition of methods to RPython classes, as done, for instance, for JavaScript [2]. However, invocation of dynamically added methods would incur a significant time penalty, though this might be mitigated by changes to the JVM or CLI, such as the recently proposed `invokedynamic` [28] instruction.

6. Related work

Jython [15] and IronPython [13] are two popular implementations of Python which target, respectively, the JVM and the CLI. Unlike our work, these projects aim to support the full dynamic semantics of Python. This results in a mismatch between the static object model offered by the virtual machine, and its dynamic Python counterpart. As an example, consider method calls: full Python allows programs may add and remove methods from classes during execution, forcing Jython and IronPython to perform the entire method look-up at runtime. In contrast, thanks to RPython's more limited semantics, we are able able to use the native instruction set of the virtual machine to perform method calls, which yields a large performance improvement.

RPython is not the first attempt to restrict a dynamic language to make it more efficient: *Slang* [12] is a restricted version of Smalltalk used to implement the Squeak virtual machine, but it is much more restricted than RPython: citing the Squeak web site [27], "Slang is essentially C with a Smalltalk syntax", while RPython is a full object oriented language.

7. Conclusion

We have presented RPython, a restricted subset of Python which is statically typed and can be compiled for the CLI and JVM platforms. The level of presentation is deliberately rather informal, since one of the main aim of this paper is showing how a statically typed object-oriented language like RPython can be quite expressive without serious runtime penalties in comparison with Java and C#.

The second main contribution of this paper concerns the development of the two back-ends for the CLI and JVM platforms. Thanks to the higher modular structure of the compiler, the two back-ends share a considerable amount of code. The benchmark results show that the RPython back-ends for the CLI and the JVM produce code which is almost as fast as that generated by the C# and Java compilers.

We have also discussed some issues both at the design and implementation level that should be addressed in the future in order to make RPython a more usable and useful

language. Moreover, since RPython was initially designed just for a specific and internal use in the context of the PyPy project, its documentation is still not adequate, especially concerning its type system, which surely deserves further insight.

Acknowledgments

The work presented in this paper fits into the PyPy project; the authors mainly worked on *ootype* and on the back-ends for CLI and JVM, and all the rest belongs to the PyPy team [30], whose help has been invaluable for getting the job done. We would also like to thank all of the anonymous reviewers for their helpful comments.

References

[1] C. S. Ananian. The static single information form. Technical Report MIT-LCS-TR-801, MIT Laboratory for Computer Science Technical Report, September 1999. Master's thesis.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, LNCS 3586, pages 428–453. Springer, 2005.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[5] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Prog. Lang. Syst.*, 28(3), May 2006.

[6] B. J. Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] A. Cuni, S. Pedroni, A. Chrigström, H. Krekel, G. Wesdorp, and C. F. Bolz. High-level backends and interpreter feature prototypes. Technical Report D12.1, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[8] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[9] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA, Geneva (CH), third edition, June 2005.

[10] C. Esterbrook. Using Mix-ins with Python. http://www.linuxjournal.com/article/4540.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley Professional, —c1995, 1995.

[12] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[13] IronPython. http://www.codeplex.com/IronPython.

[14] JRuby. http://jruby.codehaus.org/.

[15] Jython. http://www.jython.org/.

[16] S. Keene. *CLOS and the Meta Object Protocol*. Addison Wesley Publishing Company, 1989.

[17] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk: vol. 1 and 2*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[20] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[21] Martin Richards. Bcpl benchmark. http://www.cl.cam.ac.uk/~mr10/Bench.html.

[22] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[23] Microsoft .NET. http://www.microsoft.com/net/.

[24] Rhino. http://www.mozilla.org/rhino/.

[25] A. Rigo, M. Hudson, and S. Pedroni. Compiling dynamic language implementations. Technical Report D05.1, PyPy Consortium, 2005. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[26] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Companion*, pages 944–953, 2006.

[27] Slang. http://wiki.squeak.org/squeak/2267.

[28] Sun Microsystems. JSR 292: Supporting dynamically typed languages on the Java platform. http://jcp.org/en/jsr/detail?id=292.

[29] The Mono Project. http://www.mono-project.com.

[30] C. to PyPy. http://codespeak.net/pypy/dist/pypy/doc/contributor.html.

[31] G. Van Rossum. Unifying types and classes in Python 2.2. http://www.python.org/download/releases/2.2.3/descrintro/.