IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

High-Level Backends and Interpreter Feature Prototypes

Due date of deliverable: March 2007

Actual Submission date: 12th March 2007

Start date of Project: 1st December 2004 Duration: 28 months

Lead Contractor of this WP: Strakt

Authors: Antonio Cuni (HHU), Samuele Pedroni / Anders Chrigström (Strakt), Holger Krekel / Guido Wesdorp / Carl Friedrich Bolz (merlinux)

Revision: Interim

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
2007-30-01	Antonio Cuni	First draft of the High-level backend section
2007-02-26	Antonio Cuni	more about GenCLI, CLR
2007-02-26	Pedroni/Krekel	exec summary and report outline
2007-02-27	Niko Matsakis	JVM backend and backend refinements
2007-02-27	Guido Wesdorp	Draft of the JS backend section
2007-02-27	Samuele Pedroni	Security section
2007-03-08	Samuele Pedroni	Executive Summary Draft
2007-03-11	Anders Chrigström	Transparent Proxy examples, added some docs
2007-03-11	Pedroni/Bolz	Refining Transparent Proxies, Abstract, Exec
2007-03-11	Holger Krekel	Persistence, Exec Summary, Restructuring, Review
2007-03-12	Carl Friedrich Bolz	Fixes, Published Interim Version on website

Abstract

Since the beginning, the goals of PyPy's architecture and translation tool-chain included platform reach beyond Hardware/OS platforms and flexibility. The IT landscape of today encompasses both operating system/hardware platforms (such as Windows/x86 and OS X/PowerPC) and virtual machine (VM) platforms (such as the Java Virtual Machine and Common Lanugage Runtime). Languages and tools that can cover the entire range of such platforms have an advantage. We present results of validating the aspects of PyPy's translation and interpreter architecture relating to flexibility - focussing on the implementation of desirable features that usually require pervasive changes in an interpreter. The high-level backends as well as prototypes and examples relating to security, distribution and persistence features show that PyPy is well capable of providing language level features useful for middleware development.

Contents

Appendices

1. Standard Object Space documentation from the web page

1 Executive Summary

The contemporary information technology landscape includes hardware and OS platforms, but also virtual machine based platforms (JVM, CLI) and modern web browsers which are becoming a client platform of their own through JavaScript. Languages and tools that can cover all this range of platforms have an advantage.

Since the beginning, PyPy's architecture and the level of detail of our Python definition written in Python were chosen so that target platforms would not be limited to any particular hardware/OS platform. Our work on a CLI backend, which is now able to translate all of the Python interpreter, validates these initial choices. A preliminary JVM backend can already translate large examples. In the same context we explored an experimental JavaScript backend. The goal here is not to translate a PyPy interpreter but to produce dynamic web applications. Browser client code can be written in our static subset of Python to produce JavaScript, examples are available online and received community attention with the 0.99 PyPy release. All these backends share a large section of common support code that we wrote for targeting higher level (object based) platforms instead of low-level ones.

Beyond running on a given platform it is important to have access to platform specific libraries. We have successfully experimented with integration of the CLR libraries with the CLI backend, and the foundation of such access has been laid.

Together with platform reach and coverage, flexibility is a major design goal of our interpreter and translation architecture. Traditionally, many desirable features in a conventional interpreter architecture require pervasive manual changes. PyPy implements an approach that keep the core interpreter implementation mostly free of lower level implementation choices by shifting as many of these as possible to the translation tool chain. Languages and extensions can thus be implemented on a more conceptual higher level.

To validate our choices we have experimented both in adding information flow security features, which require control on all user-visible operations (through a prototype Taint Object Space). We also worked on adding language level support for transparent proxies, usable for example to make remote objects appear as local ones or to provide orthogonal persistence. In all cases, pervasive changes were not required, and prototype features could be implemented within a few hundred lines of code, employing a quick test driven development cycle.

Both the success of translating large programs to diverse platforms and the flexibility of the core Python interpreter, show that PyPy is growing and evolving into a framework for implementing dynamic languages and generating virtual machines for them. To further underline this point, there are now two prototypes of a Prolog and a JavaScript interpreter based on PyPy technology.

Confronted with wanting to target multiple platforms, the various choices for implementation details (memory management, concurrency, etc. (D07.1)) with their pervasive consequences and how to implement advanced transparent features, language implementors may want to take a serious look at PyPy, at least for inspiration. Moreover, application developers may experiment with the features and backends discussed here, examples and introductions are available online.

2 Introduction

2.1 Purpose of this Document

This document describes and assesses PyPy's portability to various non-Hardware platforms such as the CLR and the JVM. Backends we wrote to target these platform are examined. In addition it examines PyPy's flexibility by showcasing several interpreter prototypes: an information flow security prototype, and a distribution prototype and persistence proof-of-concept. This work was carried out to validate PyPy's design choices.

2.2 Scope of this Document

We present assessments and state high level benefits, results and outlooks on the emerging PyPy platform for implementing dynamic languages, based on our validation work and prototypes. For more detailed information we refer to the developer documentation and further reading.

2.3 Related Documents

- D05.1: Compiling Dynamic Language Implementations
- D07.1 Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects
- D04.2: Complete Python implementation running on top of CPython
- D13.1: Integration and Configuration

3 High-level backends

3.1 Introduction

As described in (VMC) and (D05.1), code generation is the last step of the translation toolchain and it is implemented by the backends. The first two backends available in PyPy were for C and LLVM: both these platforms are very low level and have to deal with constructs such as structures and pointers. Next, PyPy gained support for high level backends for .NET/CLI, Java and JavaScript/Browser environments.

PyPy's high-level backends are not the first attempt to implement Python in a virtual machine based environment: in particular, Jython (JYTHON-WEB), and IronPython (IRONPYTHON) are implementations of Python for the Java and .NET environments, respectively.

The approaches of Jython and IronPython are somewhat similar: they translate Python programs into native bytecode which will be execute by the hosting virtual machine, and they delegate most of the "Python-logic" to a run-time environment written in the standard language for that platform (Java and C#, respectively). For example, the Python "+" operator is translated to a call to a special method of the run-time environment that selects what concrete operation to apply (e.g., addition for numbers, concatenation for strings, etc.). These run-time environments are the rough equivalent of PyPy' *standard object space* (Appendix 1 and (D04.2)).

PyPy's approach is different: is compiles source files into Python bytecode which will be executed by the PyPy virtual machine, instead of producing native bytecode as Jython and IronPython do. This extra level of indirection results in a speed penalty (see Performance comparisons), but also gives a lot more of flexibility and possibilities that the other two implementations can't offer by design: for example, it is possible to produce Python interpreters using different object spaces than the default such as those described in this document.

Moreover, Jython and IronPython provide good integration with the hosting environment, and they allow the programmer to use the Java/.NET libraries from within Python. As discussed in the external libraries section this kind of integration is not completely supported in PyPy yet, but it will be in the near future.

In conclusion, mapping RPython (the PyPy language implementation subset of Python) to both the CLI and the JVM is a relatively straight-forward process. This is a direct consequence of RPython's design, which attempts to find a subset of Python's features that are shared by most object-oriented languages. For those times when the mapping between RPython and the CLI/JVM are non-obvious, the resulting workarounds were generally simple.

3.2 Motivation for OOType

In comparison with the low-level backends, high-level backends target platforms that use much richer, more abstract primitives. A typical high-level platform would include built-in support for such primitives as:

- classes and instances
- automatic garbage collection
- rich type system: String, List, Dictionaries, etc.
- exception handling

The best way to implement RPython constructs for the target platform is to retain the natural mapping between the two as much as possible. For example, we want to translate each RPython class into a corresponding native class, and we want to use the native built-in types for strings and lists to implement the equivalent RPython types.

The "low level type system" used by default by the RTyper (described in (D05.1)) is not suitable for this purpose because it loses too much information during the translation of the annotated flow graphs. For example, the RPython type "list of integers" is roughly represented by this structure, using C-like syntax:

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

Clearly, apart from the name of the structure there is no reference that the original RPython type was a list, and this would make the life of backends much harder.

The solution is to implement a new "object oriented type system" (OOType) for the RTyper using building blocks that can be straightforwardly translated by the backends.

The two type systems share the implementation of basic arithmetic types such as `Signed`, `Bool`, and so on. For more complex types OOType provides primitives much closer to the object oriented world instead of the structures, pointers and arrays provided by LLType. A partial list of these primitives includes:

- `Class` (for passing around classes as first-order values)
- `Instance`
- `Record` (a "struct-like" class without methods)
- `String`
- `List` (for homogeneous lists)
- `Dict` (for homogeneous dictionaries)

OOType has been designed to be usable by a variety of high level backend, including Smalltalk, JavaScript, Java and .NET. All these platforms offer a quite different object model: thus, OOType's object model is designed in a way that all its features are either available natively in a large number of target platforms or easy implementable without loss of efficiency.

As an example, consider having functions and classes that are first-order entities: backends for platforms such as Java (and partially .NET) that have no built-in support for this can easily and efficiently simulate it by wrapping such entities into a class. On the other hand, it's impossible to efficiently support multiple inheritance if the platform supports only single inheritance.

As a result, the object model implemented by OOType is quite Java-like. The following is a list of key features of the OOType object model which have a direct correspondence with the Java or .NET object model:

- classes have a static set of strongly typed methods and attributes;
- methods can be overridden in subclasses; every method is "virtual" (i.e., can be overridden); methods can be "abstract" (i.e., need to be overridden in subclasses);
- classes support single inheritance; all classes inherit directly or indirectly from the ROOT class;
- there is some support for method overloading. This feature is not used by the RTyper itself because RPython doesn't support method overloading, but it is used by the GenCLI backend for offering access to the native .NET libraries (see the external libraries section);
- classes, attributes and methods don't have access restrictions (this could be mapped into making everything public or for example in Java package private): OOType is only used internally by the translator, so there is no need to enforce accessibility rules. A future separation into public vs. internal would be also possible.
- classes and functions are first-order entities: this feature can be easily simulated by backends for platforms on which this is not a native feature;
- there is a set of built-in types offering standard features: List, Dict, DictItemsIterator, CustomDict, String, StringBuilder. For most of them the semantics are obvious, for instance CustomDict behaves like a Dict which you can pass custom hash and equal functions to;
- List, Dict, DictItemsIterator and CustomDict are parametrized by type: this allow backends such as GenCLI to exploit native support for generic programming; backends for platforms like Java need to simulate it using type erasure.

The resulting object model is very convenient from the RTyper point of view, because the RPython object model is also somewhat Java-like: so, for example, RPython classes will be mapped to OOType classes by the RTyper, and after that to native classes by the backend.

3.3 CLI/.NET backend

GenCLI is the most mature high level backend so far. It targets the CLI Virtual Machine (ECMA-335), which stands at the heart of the .NET Framework, paying particular attention for being compatible with its two most widespread implementations: Microsoft CLR (CLR) and Mono (Mono). It was born as a part of Antonio Cuni's master thesis (PYDOTNET), then it has been further developed.

3.3.1 Implementation Notes

Being a high level backend, GenCLI uses the object oriented type system of the RTyper described above. The mapping of most OOType features is straightforward because the CLI virtual machine natively supports them:

- all primitive types such as `Signed` and `Float` has a direct correspondent;
- built-in parametric types such as `List` and `Dict` can easily be mapped to their .NET *generic* equivalents `System.Collections.Generic.List` and `System.Collections.Gener`
- RPython classes are mapped to CLI classes: as we saw above the OOType object model is very similar to the CLI one, so no special care is needed;

Still, there are features that need special treatment by the backend because either CLI doesn't support them or it supports them differently.

In addition to `Dict`, OOType also provides the `CustomDict` type, whose constructor takes two functions used for computing key equality and hash values. The corresponding .NET class `System.Collections.Generic.Dictionary`, however, takes an instance of a class implementing the `IEqualityComparer` interface instead of two distinct functions: to fill the gap between the two signatures GenCLI generates a new class implementing that interface for each distinct pair of equality and hashing function and then passess an instance of this to the `Dictionary` constructor.

A similar trick is used for implementing functions as first-order objects: since .NET doesn't directly support those, we need to wrap each function we want to pass around into a *delegate*, which is no more than an instance of a subclass of `System.MulticastDelegate` providing the `Invoke` method with the correct signature: since we need a different delegate type for each different signature, GenCLI keeps track of every function we want to use as a first-order object, then generates the right delegate types depending on their signatures.

Finally, RPython also supports classes as first-order objects, while CLI doesn't: to solve the problem we can pass around an instance of `System.Type` instead of the class itself, that is automatically created by the virtual machine for each class we declare in our assemblies. Then we can dynamically create an instance of that class by calling the `RuntimeNew` helper method, 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]);
}
```

The target language of GenCLI is IL, the standard CLI assembler; the generated code is fed to `ilasm` to produce the real executable.

Choosing IL as the output language has been very convenient for a number of reasons: despite being an assembler-level language it gives access to all .NET native constructs such as classes, interfaces, custom attributes, etc.; moreover the classes and methods don't have to be declared in a particular order because `ilasm` itself takes care of resolving dependencies. GenCLI produces the IL code by starting from en entry-point and recursively traversing the call tree, emitting methods and classes as they are found, which is considerably simpler than the approach of GenC of collecting all functions and types before emitting them (D05.1).

3.3.2 External libraries

One of the goals of having a high level backend is to let the programmer access the outside environment from within Python as transparently as possible. In this section we describe how it is possible to use classes and methods of the .NET framework using GenCLI, which is currently the most advanced high level backend. We can access .NET libraries both at the RPython level and at the application level. The RPython level is not really meant to be complete or very usable: it only provides what we need to build the application level bridge.

Both PyPy and .NET uses a similar technique to handle different kinds of objects in an uniform way: they design a class hierarchy with a known interface and enclose each kind of object into a different subclass of that hierarchy. To avoid confusion, we use the terms *wrapping* (D04.2) when referring to PyPy and *boxing* when referring to .NET.

3.3.3 RPython-level bindings

Adding external library support to RPython is not straightforward because we have to convince the annotator ((D05.1)) that some classes/methods don't really exist but are just placeholder for external entities that will be replaced by the backend. Moreover there are some .NET features that are used by external libraries but not directly supported by RPython such as method overloading.

To annotate external function calls we need to know their signature in order to get the result type given the types of the arguments. The simplest way would have been to manually write the signature of each method we want to use, but it would have been time consuming and error-prone. Instead, we use an external tool written in C# that uses .NET reflection to retrieve signatures and other information of classes we want to use: this information is then automatically saved to disk to improve performance of the next translations.

Thanks to the modular design of the translation toolchain the additions to the annotator and RTyper consist of only a few hundred lines contained in a single file. Once the flow graphs are annotated and rtyped correctly it's easy for GenCLI to give access to external functions, because it only needs to map the placeholders for types and methods to their real .NET name.

3.3.4 The `clr` module

The RPython bindings for .NET are completely static, because the references are resolved at translation time, as a C# compiler would do. By contrast the application-level bindings are completely dynamic by making heavy use of reflection: they are provided by the built-in `clr` module.

As of the rest of PyPy, also the `clr` module is stratified into several levels, each one providing more advanced features and relying of those of the lower ones:

- the lowermost level contains the RPython static bindings described above;
- the *Python/CLI bridge* level let you to instantiate and manipulate .NET objects at a very low level.
- finally, application-level bindings package the functionality provided by the Python/CLI bridge in a pythonic way to let you create and manipulate .NET objects and classes as they were native Python ones.

Dynamic app-level bindings
Python/CLI bridge
Static RPython-level bindings

3.3.5 Python/CLI bridge

The goal of the *Python/CLI bridge* is to convert objects when they cross the boundaries between the Python world and the .NET world. There are several kinds of conversion, depending on the type of the object and on the direction we are going.

For primitive Python types having a direct correspondent in .NET we try to preserve the intuitive semantics as much as possible: this means for example that Python integers are automatically converted to (boxed) .NET integers, and vice-versa. As a special case, the Python value `None` is converted to the .NET *null value*.

Objects of non-primitive types are handled differently: since their type doesn't exists in the other world, they can't be directly manipulated once they passed the border; therefore the only possible use-case is to pass them as black boxes to be retrieved later, for example as elements of some collection. The concrete conversion applied depends on which direction we crossing the border.

Passing Python object to .NET is straightforward, because internally they are already implemented as instances of a .NET class automatically generated by GenCLI during the translation: all we need to do is to cast it to `System.Object`.

By contrast, to pass .NET objects to Python we need to wrap them so that they can be manipulated by the Python interpreter.

Moreover, special care is paid to detect objects that already belongs to the world they are going to: for example, when we retrieve a Python object that we put into a .NET collection we don't want to wrap it as it were a .NET object but we can just let it pass through because it is already a wrapped object. Similarly, when passing from Python to .NET we want to unwrap .NET objects that were wrapped earlier.

Once we have got a .NET wrapped object, we can call native .NET methods by invoking `call_method`, which takes the method name as a string and the list of actual arguments; the arguments as well as the return value are automatically converted as described above. Then if the given method is overloaded the best match (if any) is taken and invoked. If the method doesn't exists or it's impossible to determine a best match, an exception is raised.

Finally, the Python/CLI bridge also provides functions to create new .NET wrapped objects given the name of the class, and to invoke static methods given the name of the class and the name of the methods.

Internally all the operations are implemented using the RPython static bindings to .NET and heavily relies on reflection to achieve their goals. As an example, consider the implementation of `call_method`: it first gets the .NET type of the target object, then looks up the method descriptor corresponding to the given name and types of parameters, then invoke it.

This mechanism provides a simple and low level interface to manipulate .NET objects from within Python: this solution is similar to the one adopted by Kawa (KAWA) to let Scheme programs access the Java platforms.

However we think that this mechanism is still too low-level for being really usable by the end user, so its only goal is to provide enough functionality for the application-level bindings to work.

3.3.6 Application-level bindings

Application-level bindings provide Pythonic access to the .NET libraries; other projects such as Jython (JYTHON) and IronPython (IRONPYTHON) have already shown that it's possible to integrate the Python object model with the one of the enclosing virtual machine.

The main entry-point of the `clr` module is the `load_cli_class` function: it takes the name of a .NET class and return a real, application-level, Python class that behaves as much as possible as a standard Python class. `load_cli_class` also makes use of reflection to discover which methods, properties and other features are supported by the .NET class in order to map them to Python.

For each .NET method the application-level class contains a Python method which is implemented by delegating the real call to the underlying Python/CLI bridge. As an example, the `Add` method of the `System.Collections.ArrayList` class might look like the following, where `__cliobj__` contains the reference to the object wrapped by the Python/CLI bridge:

```
class ArrayList:
      def Add(self, *args):
          return self.__cliobj__.call_method('Add', args)
```

Actually the real implementation is a bit different: instead of relying on Python functions for representing methods, we use our own `MethodWrapper` class which implements the descriptor protocol in a similar way as functions but returns our own classes for bound and unbound methods: it is finally up to those to do the invoke `call_method` when they are called. This is necessary because we don't know the exact name of the methods until `load_cli_class` is invoked at runtime, so there are no chances to have a method like the one presented above unless we want to use `exec`.

A similar mechanism is used to implement other important features such as static methods and properties.

Special care is taken to preserve the same syntax used by C# and other .NET languages when possible. For example, .NET *indexers* are mapped to Python `__getitem__` and `__setitem__`, so that you can call them by simply using the usual `[]` notation.

Finally, there is some support for catching exceptions throwed by .NET methods, though it is still incomplete. The main drawback is that no distinction is made about them: every .NET exception is mapped to Python `StandardError`, though it preserves the original error message.

As an example, here is an interactive session showing some of the features described:

```
>>>> import clr
>>>> ArrayList = clr.load_cli_class('System.Collections', 'ArrayList')
>>>> mylist = ArrayList()
>>>> bound_method = mylist.Add
>>>> bound_method(0)
0
>>>> bound_method('foo')
1
>>>> print mylist.Count, mylist[0], mylist[1]
2 0 foo
>>>> print mylist[2]
Traceback (most recent call last):
  File "<inline>
print mylist[2]", line 1, in <interactive>
StandardError: Index is less than 0 or more than or equal to the list count.
Parameter name: index
2
```

To summarize, here is a list of features that are actually supported:

- dynamic load of classes;
- automatic conversion between Python and .NET and vice-versa when it makes sense;
- automatic resolution of overloading;
- methods and static methods;
- properties and static properties;
- indexers;
- translation of .NET exceptions to Python exceptions (incomplete).

Being still a prototype not all the desirable features are implemented, but they will be added in the future:

- better integration of the two exception hierarchies;
- support for inheritance and subclassing;
- ability to add Python callables to .NET delegates (i.e., ability to call Python code from .NET);
- better mapping of features that have a direct correspondent; e.g., automatically turns classes that implements the `IEnumerable` interface into Python iterators.

Some of the mentioned features are not straightforward to implement because the object models of Python and .NET are very different. This topic will be part of the PhD thesis of Antonio Cuni at Università degli Studi di Genova.

3.3.7 Performance comparisons

In this section we will examine performances of the Python interpreter produced by GenCLI. Similar analysis could be performed also for others backend once they will be as complete as GenCLI.

Following is a list of all the different Python implementations that have been considered in the comparison, toghether with their version:

- CPython 2.4.4: the reference implementation; written in C;
- IronPython 1.0.1: the first implementation of Python for .NET; written in C#
- pypy-c 0.99: the PyPy interpreter produced by the C backend;
- pypy-cli 0.99: the PyPy interpreter produced by the CLI backend;

Moreover, also the .NET environment used for running pypy-cli and IronPython is important. Both have been tested with two different implementations:

- Mono (Mono) 1.2.3.1 on Linux
- Microsoft CLR 2.0 on Windows XP

pypy-cli is built over several layer of abstraction, and each of them could affect performances:

- the quality of the code generated by GenCLI, compared for example to the code generated by a C# compiler;
- the quality of the hosting virtual machine and the correspondent JIT compiler;
- the efficiency of the PyPy's objects implementation compared to CPython's and IronPython's ones.

To measure all these differences, we used a RPython implementation of the classical Martin Richards's benchmark (RICHARDS), which can be either translated using one of PyPy backends, interpreted by various Python implementations and also compiled as a built-in PyPy module, useful for factoring out the interpretation overhead.

The following tables summarizes the results; the values are expressed in milliseconds and represent the "average time per iteration" (the smaller the better). 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.

Windows benchmarks	
Implementation	**Result**
richards-c#	7.093
richards-gencli	13.312
CPython richards.py	1139.632
IronPython richards.py	1751.246
pypy-cli richards.py (built-in)	5952.501
pypy-cli richards.py	12010.541

Linux benchmarks	
Implementation	**Result**
richards-c#	4.000
richards-gencli	13.982
CPython richards.py	689.215
IronPython richards.py	1502.642
pypy-cli richards.py (built-in)	9882.231
pypy-cli richards.py	22113.572

First, it must be noted that the GenCLI vs. C# slowdown is about 3.4x under Linux/Mono but only about 1.8x under Windows/CLR: a possible interpretation of this result is that Mono JIT compiled is specifically tailored for code produced by the C# compiler, while Microsoft JIT compiler handles well also different kinds of IL code. Thus, in the following sections we will refer only to the Windows benchmarks, because the Mono-specific slowdown impacts too much on the performances.

Ideally, the code produced by GenCLI should be as fast as the code produced by the C# compiler: the slowdown is attributable to the youngness of GenCLI and will likely converge to zero in the future.

Then, when comparing pypy-cli with IronPython it must be considered their different approach, as discussed in the *related work* section: IronPython directly produces CLI bytecode, while PyPy produces Python bytecode which is then interpreted by its own virtual machine.

The overhead of the extra-level of indirection can be measured by comparing the speed of richards.py when compiled as a built-in module and when executed as a normal application-level module. As we can see, the slowdown is about 2x.

Then, we can compare the execution speed of pypy-cli and IronPython: the slowdown "out of the box" is about 6.8x, but as we saw above this is the results of three different aspects:

- the GenCLI vs. C# slowdown: about 1.8x;
- the interpretation overhead: about 2x;
- the efficiency of PyPy standard object space compared to IronPython run-time environment; this can be measured indirecly by removing the other two factors above: about 1.8x.

Finally, it must be said that there is still a lot of improvements and optimizations to be done, so in the future the gap could become less and less wide.

3.4 JVM backend

The Java Virtual Machine (JVM) backend is one of the newer backends developed for the PyPy system. Developing the JVM backend was a fairly smooth process: RPython's type model combined with the object-oriented type system proved to be a good match for the JVM. Due to the many similarities between the JVM and the CLR, it was even possible to reuse much of the CLR backend's code.

The JVM backend does not generate Java source files, but instead generates byte-code directly. It emits text files in the format expected by (Jasmin), an assembler for JVM byte-code, and then invokes Jasmin to create the binary .class files. Jasmin's serialization format is used because Sun has not defined a standard textual format for JVM byte-code. Although we emit byte-codes directly, all the examples in this section are given in Java source text for readability.

The JVM backend is not yet as fully developed as the CLI backend. It cannot translate the full Python interpreter yet. Most of the conceptual hurdles have been overcome, however, and what remains is to flesh out the corners of the current code and ensure that it handles all the corner cases correctly. The JVM backend has been used to translate rpystone, and the resulting code ran roughly 20 times slower than the corresponding C version. It is important to keep in mind, however, that the JVM version has no applied optimizations, and that the JVM backend has not been optimized for speed in any way.

One problem that may occur in future JVM backend development is that the generate JVM code could exceed the built-in limits of the JVM. JVM class files impose several arbitrary limits, such as the maximum size of a method's byte-code (64kB). Currently, all RPython methods are translated directly into a single JVM method: however, if the RPython method is too large, this technique will not work. The solution would be to subdivide the RPython method and spread it across several JVM methods. This process could be easily automated. The Static Single Information (SSI) form of the RPython flow graphs would greatly help, as there are no local variables with a scope larger than a single basic block. It is also possible that Sun will modify the class file structure to alleviate this limit.

3.4.1 Implementation notes

Overall, the JVM is a fairly simple virtual machine, and generally eschews advanced or specialized features that are not needed for executing Java; as a result, there are a few areas where the translation from RPython into JVM byte-code is not entirely straight-forward:

- unsigned integers and longs
- parametric built-in types (such as `List` and `Dict`)
- functions and classs as first-order objects
- throwing exceptions

The workarounds for each of these problems is covered in the following sub-sections.

3.4.2 Unsigned Operations

Because the JVM does not have native support for unsigned operations, the JVM backend only uses signed values internally; therefore, an unsigned int is represented as a signed int, and an unsigned long as a signed long. This is generally not important, because most operations, such as addition or bitwise manipulation, work equally well on signed and unsigned values (so long as the two are not mixed). The backend must pay careful attention when unsigned values are compared, however, because the default integer comparison functions of the JVM assumed signed values. For this purpose, special library functions were introduced to handle comparisons between unsigned integers.

3.4.3 Parametric built-in types

Prior to version 1.5, the Java language had no support for parametric types. Container classes such as `Vector` and `Hashtable` were implemented as containers of generic `Object` pointers, and explicit casts were required when objects were removed from them to assert their type. Although generics were added to Java 1.5, the JVM was never modified to match: instead, the Java compiler simply inserts the required casts from `Object` when necessary. The PyPy backend takes a similar approach: whenever a value of parametric type is returned from a function, a cast is inserted into the byte-code to convert it into its correct type. These casts might seem unnecessary, because the PyPy RTyper has already validated that the objects in the array are of the correct type, and therefore the casts can never fail; however, the casts are required to allow the JVM byte-code verification to succeed.

In addition to inserting casts around parametric types, the JVM backend must also allow for vectors of primitive types, such as signed integers or doubles. This is done by boxing: an instance of an appropriate JVM wrapper type, such as `java.lang.Integer` or `java.lang.Double`, is created and inserted into the parametric container class; when the value is later removed from the container, it is un-boxed.

3.4.4 Pointers to Functions and Classes

To handle functions as first-order objects, the JVM must create a class for every function passed around. This class contains a single method "invoke", which executes the function. In addition, for every different signature of function pointer, an abstract base class is generated with the same "invoke" method. This allows a variable to potentially point at one of a number of different functions, and uses the built-in virtual dispatching of the JVM to handle the actual call.

As an example, imagine that there are two functions, each of which takes a single signed integer argument and returns a signed integer result: `negative`, which multiples the argument by -1, and `half`, which divides it by two. The JVM backend would then create three classes in all: one abstract base class, and two concrete implementations thereof:

```
abstract class BaseClass {
  public abstract int invoke(int arg);
}

class negative_delegate extends BaseClass {
  public int invoke(int arg) {
    return -arg;
  }
}

class half_delegate extends BaseClass {
  public int invoke(int arg) {
    return arg/2;
  }
}
```

This way, a pointer to the function `negative` can be obtained by instantiating the `negative_delegate` class.

As GenCLI does, classes as first-order objects can be easily handled using the JVM's reflective capabilities. A pointer to a class becomes a pointer to an instance of `java.lang.Class`, and any attempt to instantiate this pointer simply requires a small amount of reflection.

3.4.5 RPython Exceptions

Mapping RPython exceptions to the JVM also requires special handling. The problem is that the JVM only allows subclasses of `java.lang.Throwable` to be thrown as exceptions; however, the translation process currently translates RPython exception classes into normal Java classes that do not derive from `Throwable`, which causes byte-code verification errors when attempting to throw them. There are a number of possible solutions; currently, the JVM backend uses the most expedient, which is simply to create a wrapper around exception instances when trying to throw them. The result looks like:

```
throw new JvmExceptionWrapper(pypyExceptionInstance);
```

The problem with this solution is that it does not allow the use of the JVM's built-in mechanisms to select exceptions by class. Normally, in order to catch an instance of SomePyPyExceptionClass, one can use a simple catch block like so:

```
try {
  ...
} catch (SomePyPyExceptionClass exc) {
  ...
}
```

When using a wrapper, however, all exceptions will be of type JvmExceptionWrapper; therefore, the `catch` construct above must be converted into:

```
try {
  ...
} catch (JvmExceptionWrapper _exc) {
  if (!(_exc.wrapped instanceof SomePyPyExceptionClass))
    throw exc;
  SomePyPyExceptionClass exc = (SomePyPyExceptionClass)_exc.wrapped;
  ...
}
```

There are a number of possible improvements to this system. One obvious solution is to map the RPython `Exception` class to `java.lang.Throwable`. Another is to avoid the JVM's exception mechanism altogether, and instead indicate thrown exceptions using return values; this would be advantageous because exceptions in RPython are used more frequently than in the JVM, and a lighter-weight mechanism may be appropriate.

3.5 JavaScript backend

JavaScript (formally known as ECMAScript, (ECMA-262)) is implemented in every modern browser and has grown to be the language of choice for client-side programming. However, since JS lacks some important features that Python does provide, the JS backend is expected to become widely used in the near future.

Unlike the CLI and JVM backends, the JavaScript backend does not aim at running a full Python interpreter, but rather at translating RPython programs into JavaScript and running them on a browser. It is an approach worth of experimentation compared to ad-hoc AST transformation approaches that basically map to some parts of JavaScript from Python (or other languages) surface syntax.

The JavaScript backend has a particular focus on enabling quick testing cycles. By contrast, it today is difficult to automatically test JavaScript applications, because stand-alone JavaScript interpreters do not provide the same API as browsers, they do not implement a DOM (URL: http://www.w3.org/TR/DOM-Level-2-Core/), nor the XMLHttpRequest that is used to interact with the server in AJAX applications. Additionally, there are no established practises regarding testing comparable to the ones in the Python world.

3.5.1 Implementation notes

One of the most important reasons to write a JavaScript backend is that testing JavaScript is hard and not commonly done. RPython code written for translation is easy to test, and can use the comparatively advanced testing tools that are available to CPython. On top of that, there are no restrictions in the written code: the core browser API is made available as a CPython module, and additional modules are relatively easy to add.

Our approach tries to solve the testing problems by transparently wrapping the server-side API: method calls to functionality provided by the server are rendered as XMLHttpRequest calls, provided that input and output types are known in advance. Our framework then calls the appropriate method on the server, serializing the types automatically. We provide a simple wrapper to have similar behaviour in CPython, allowing us to run our full program on a normal CPython interpreter.

Because JavaScript supports a prototype-based model rather than class-based object orientation, implementing object orientation on this backend was not as straight-forward as on the other high-level ones. To overcome this, we implemented an `inherit` function, which takes a type and copies all the methods down the chain at class creation time. Other class members are copied in a similar way, but at object instantiation time. This does not break consistency compared to the other backends since RPython does not allow changing base class members at runtime anyway.

To allow access to external APIs such as the browser DOM, XMLHttpRequest and JavaScript libraries such as Mochikit, we used an external object description mechanism we call `BasicExternal`. To expose a JavaScript API, a user will need to describe all external methods and types used on an instance of the `BasicExternal` class. The information on this instance is used by the annotator and RTyper during RPython translation. This means the real methods are never seen by the annotator, just knowing about the type information and method signatures is enough for translation purposes.

4 Interpreter Prototypes and Extensions

In this section we discuss prototypes and approaches that make use of PyPy's Interpreter architecture (VMC) and in particular the strict separation of bytecode evaluation (including control flow) and performing operations on application level objects. Making use and extending the interpreter architecture can be done rather independently of translation aspects, which makes that all of the presented approaches can work equally well with our lower level and higher level backends.

4.1 Security

4.1.1 Motivation and Related Work

Capability-based security and language support for information flow control are current vital and most interesting areas of research on language-level security (RC), (MLS).

Capability-based security aims at allowing to safely construct systems out of trusted and untrusted components by enabling a consistent application of the principle of least authority. Current designs use objects as capabilities and encapsulation to realise this, capabilities should only be obtainable through the graph of object to object references and message sends.

Most practical languages have forms of implicitly shared global state or capabilities, or avoid this by implementing loading and importing mechanisms. This implies a careful language design or redesign. Since Python doesn't support any form of encapsulation, has widespread reflective functionality and for example always shares available builtins, it's not directly suited for capability-based security. Implementing a capability-based security prototype would not have fitted PyPy's focus, since it would have mostly required working at the language design, and not so much at implementation level.

Information flow control labels sensitive data and makes sure that this data is not accessible without enough authorization. For a system run on behalf of principals with different authorization levels or composed of parts each run with different authorization levels, it must be ensured that no sensitive data leaks through the data-flow in non-authorized parts or to non-authorized principals.

Approaches based on language design and static analysis of the propagation of data with their labels through the data-flow of programs have been researched to make such pervasive checks viable (for example see (JFLOW)). More recently a research effort has been started under Michael Franz to integrate support for such labeling and checks directly in modern Java virtual machine designs, in ways such that unmodified programs can be run with orthogonally specified security policies. Some extent of load-time analysis and the help of dynamic compilation to reduce the overhead play a role in this latter approach (MLS).

In the context of PyPy and Work Package 12 we implemented some of the essential ideas of such approach at a prototype scale in the form of an object space (D04.2) we call the *Taint Object Space*. The Taint Object Space directly implements the simple example policy in (MLS), although generalisations to multi-levels security lattices and orthogonal policies are easily conceivable.

On the 5th of February 2007, a workshop was organized at IBM Zurich Labs, by Matthias Schunter, with Michael Franz himself in which we could discuss and validate the prototype approach and present it in the larger context of PyPy architecture and flexibility goals. During the workshop we largely received positive feedback, and were able to discuss aspects that the prototype only partially addresses, like what to do if the control flow itself depends on sensitive information: the conclusion was that this - in the absence of whole program analysis - is still an open research question in the field.

4.1.2 Taint Propagation

The prototype showcases PyPy's flexibility: the security checks could be introduced by wrapping the Standard Object Space in a proxying object space - the Taint Object Space - which uses delegation and is situated between the Standard Interpreter and Standard Object Space. Both are otherwise left untouched and the latter is still responsible for the normal execution of operations. See (D04.2) for details on the PyPy architecture and object spaces.

The Taint Object Space implements a simple dual-level security model: objects are either regular (untainted), or sensitive (tainted).

In practice classifying objects as tainted may be useful in two cases: for secret data, or for untrusted data. In the former case, tainting the secret data prevents unrelated parts of the program from accidentally leaking it outside the program to a malicious observer. In the latter case, tainting makes sense for example for untrusted input from a web form, to ensure that only a small number of places in a program need review to ascertain that they are manipulating it as carefully as required. For example, no database query should be assembled from any untrusted, unchecked string - a classical CGI script attack is to send malformed inputs to trick the script into constructing a different query than it intended to. This is very similar in application to Perl tainted mode.

Tainted objects are implemented as a wrapper around regular objects. A builtin function `taint` allows to wrap a regular object (of any type) in a tainted wrapper. Declassification is invoked by an `untaint` builtin function - it is the only way to inspect tainted objects. A program can be considered secure in its manipulation of secret information if all the places that use `untaint` carefully prevent the untainted object from leaking elsewhere in the program; it can be considered secure against untrusted inputs if all the results of `untaint` are carefully checked before being used. The point of the approach is that these two properties can be checked locally, around each call to `untaint`, and do not require a whole-program review.

Tainted objects cannot be inspected, but they can still be safely operated on. All object space operations will produce a tainted result if at least one argument is tainted. If they fail, a so-called tainted bomb is created: this is a tainted-like object which will raise an exception when the program tries to declassify it. The exception is not raised immediately, because its mere presence (as well as the error message) would be a potential information leak by itself.

Unboxing operations, which are e.g triggered for output and testing the truth value of a tainted object, always result in an error. This means that declassification is needed before output.

```
>>>> from pypymagic import taint, untaint
>>>> a =  taint(3)
>>>> b = a + 2
>>>> b
Traceback (most recent call last):
  File "<console>", line 1, in <interactive>
TaintError
>>>> untaint(int, b)
5
>>>>
```

This also means control flow cannot depend on tainted values. To help structure code that needs to manipulate tainted values, and to avoid scattering declassification operations everywhere, a higher-order builtin function is provided: `taint_atomic`, which constructs out of a function a new function that behaves "atomically" with respect to tainting, like the space operations themselves, in the sense that if at least one of the arguments is tainted the result will be put in a tainted wrapper too:

```
>>>> def myabs(x):
....     while x < 0:
....         x = -x
....     return x
....
>>>> myabs = taint_atomic(myabs)
>>>> myabs(-8)
8
>>>> z = myabs(taint(-42))
>>>> z
TaintError
>>>> untaint(int, z)
42
```

Because of PyPy's clean separation and interface between interpreter loop and object spaces (D04.2), the Taint Object Space's implementation is straightforward. All operation implementations in the Taint Object Space generically check whether some arguments are tainted wrappers. In case they are, their unwrapped contents are passed to the Standard Object

Space implementation of the operation, to which the regular execution is delegated. The regular implementation will return a regular object. If at least one argument was tainted the result will be taint-wrapped before being returned to the caller of the operation.

4.1.3 Conclusion

In our architecture the object space is ultimately responsible for all object manipulations, so by construction it is not possible for user programs to circumvent taint-wrapping, or for accidental declassification to occur.

A PyPy translated to C with the Taint Object Space is around 33% slower than with the non-proxied Standard Object Space. This is with the most straightforward implementation: by letting the proxied space access some of its original functionality unproxied in a controlled way it should be possible to reduce the overhead further.

We experimented writing a small SQL injection demo, in the spirit of the application examples referred in (MLS). Closing the holes using the Taint Object Space turned out to be easy.

The Taint Object Space implementation is only around 300 lines of code, using typical Python meta-programming techniques. Reducing the overhead and further improving our internal interfaces for better proxying would require only slightly more effort. We think the Taint Object Space validates our flexibility goals.

4.2 Transparent Proxies, Distribution and Persistence

4.2.1 Introduction

In languages such as Smalltalk is it relatively easy to implement general proxy objects to be used for example for remote object access. Python does not provide a simple uniform object model solely based on dispatching with a `doesNotUnderstand` form of fallback like Smalltalk does. Instead, a more complex attribute-oriented model is used, in which so called descriptors live in type namespaces and are "bound" on lookup producing method or property-like behavior. Detailed descriptions of this model can be found in (PYTHONTYPES), (PEP252) and (PEP253).

On top of this, Python includes a set of builtin types - mutable containers like lists and dictionaries but also execution frames and tracebacks. These builtin types participate in the model with opaque descriptors in their namespace. Furthermore, both internal and user code sometimes rely on `isinstance` or exact type checks.

For these reasons, it is quite challenging to implement the sort of proxying required for example for some object oriented approaches to distribution, especially if features like uniform access to local and remote objects, or for example remote tracebacks inspectable as local ones, are to be included. Even the possibility of subclassing such builtin types does not help here, since most interpreter-internal operations access the inherited and opaque part of the object. Not all builtin types are subclassable or their behavior if some methods are overridden is not specified.

To overcome this problem, we introduced *transparent proxies*. These are objects that masquerade as instances of a specific type, possibly builtin, but with completely customizable behaviour. Their behaviour is determined by a controller function, which is called whenever an operation is performed on the proxy instance. Transparent proxies are constructed specifying a controller and a type.

A simple illustration of using a Transparent Proxy:

```
$ py.py --with-transparent-proxy
>>>> from pypymagic import transparent_proxy
>>>> def f(operation, *args, **kwargs):
....    if operation == ’__getitem__’:
....         return 42
....    raise TypeError
....
>>>> l = transparent_proxy(list, f) # Create a list object that
                                    # forwards all operations on it
                                    # to the function f

>>>> type(l)    # Looks like a list
<type ’list’>
>>>> l[3]      # This calls f with ’__getitem__’ as first argument
42
```

A more advanced example is a list that will remember what operations have been performed on it:

```
from pypymagic import transparent_proxy, get_transparent_controller
from types import MethodType

class ListController(object):
    def __init__(self, l):
        assert isinstance(l, list)
        self.l = l
        self.history = []
        self.proxy = transparent_proxy(list, self.perform)

    def perform(self, operation, *args, **kwargs):
        self.history.append(operation) # remember the operation performed
        # perform the operation on the proxied object
        result = getattr(self.l, operation)(*args, **kwargs)
        if result is self.l:
            # If the result is the proxied list
            # return the proxy instead.
            result = self.proxy
        elif (isinstance(result, MethodType) and
              result.im_self is self.l):
            # Convert methods bound to the proxied list
            # to methods bound to the proxy.
            # This is to have calls to the method become calls
            # to perform.
            result = MethodType(result.im_func, self.proxy, result.im_class)
        return result
```

With this we can do:

```
>>>> l = []
>>>> c = ListController(l)
>>>> lst = c.proxy # Get the proxy object
```

```
>>>> lst += [1, 2] # This will call perform with ’__iadd__’ as operation
>>>> lst.append(3) # This is really two operations, first a
                   # __getattribute__ to get the append method and
                   # then a call to the method.
>>>> len(lst)      # This is a __len__ operation
3
>>>> l             # The operations have been performed on the proxied list
[1, 2, 3]
>>>> c.history     # We can check what operations have been performed.
[’__iadd__’, ’__getattribute__’, ’append’, ’__len__’]
```

Microsoft .NET framework has a related `TransparentProxy` class, used in their remote object access implementation. A `TransparentProxy` there can also masquerade as an instance of some class or pretend to implement a set of interfaces. All dispatching and object access mechanisms in the VM are made aware of `TransparentProxies` (REMOTING), (TP). All these operations are then forwarded to a `RealProxy` which can implement it in any way (very similar to what our controller functions do).

In our case we exploited PyPy's design and flexibility as much as possible to implement the transparent proxy feature without pervasive changes to the bytecode implementation or the interpreter loop. Our current prototype focuses on allowing the construction of transparent proxies for builtin types: in PyPy, the Standard Object Space allows for multiple implementations of each user-level builtin type, supported by *multimethods* and their so-called *slicing* (see Appendix 1). For example, transparent proxies of lists are written as a new implementation for the `list` Python type. For "internal" types not provided by the Object Space, like frames, we used a more traditional type-checking and fallback mechanism.

This involved adding less than 300 lines of code. Transparent proxies are completely optional and separated from the interpreter and can be enabled at translation time (D13.1). Perfomance-wise they have no significant impact on the rest of the application.

4.2.2 Distribution using Transparent Proxies

To test and showcase transparent proxies we wrote a prototype library for object-oriented distributed computation. The `distributed` library is an attempt to provide transparent, lazy access to remote objects. This is accomplished with application level code, so as a pure Python module.

The implementation uses an RPC-like protocol. Essentially only immutable atomic objects (integers, strings etc) are transferred by value, otherwise transparent proxies encapsulating remote references are used. Operations are either delegated to the remote side, or some internal data is temporarily transferred and the operation performed locally. This means it does not rely on objects being pickleable, nor on having the same source code available on both sides.

Using this library it is possible to debug a remote exception with a debugger on the local side, having a transparent proxy referencing the remote traceback. This shows how well transparent proxies can behave like built-in interpreter objects - even the debugger works normally on them. Indeed, it is very hard to distinguish a remote object from a local one.

Compared to other remote object libraries for Python, the possibility of having remote objects appear as normal local Python objects is the decisive factor. Neither Pyro (PYRO) nor Twisted Spread (SPREAD), two commonly used Python distribution libraries, can distribute instances of builtin types such as lists, dicts or frames and tracebacks. Instances of user-defined classes even have to subclass a specific class to make them distributable.

4.2.3 Orthogonal Persistence and Transparent Proxies

To implement orthogonal persistence in a language, three basic principles are of importance (OP):

- Persistence Independence: Programs look the same whether they manipulate short-term or long-term data.
- Data Type Orthogonality: All data objects should be allowed full range of persistence irrespective of their type.
- Persistence Identification: The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system.

One strategy to implement language level persistence is based on "suspended computations" and "reachability of objects", usually providing "transient" markers to cut the reachability graph of persisted objects (OPJ). PyPy's thread pickling and cloning facilities provide the core functionality to implement such schemes (VMC) and these primitive features themselves validate PyPy's translation architecture capabilities. However, persisting and checkpointing whole computations usually implies a need to care for restricting the reachability graph, a potentially unwelcome overhead for an application programmer, arguably compromising the idea of orthogonal persistence.

We believe that providing selective means to transparently persist all objects should be more interesting for application programmers. In that case, transparency means that language level objects appear as completely regular objects ("Persistence Independence" and "Data Type Orthogonality"), although in fact they are persisted to an application specified storage. Using particular storage mechanisms, choosing databases and policies are left to the programmer of a particular "support system" (see "Integration Concepts" in (OP)) and is to be orthogonal to application development.

Transparent Proxies are a core building block for providing such control over objects manipulations. The support system code, which is typically a small part of the application, may intercept changes to all objects without requiring intrusive changes like extra attributes or proxy types that would be visible to the rest of the application.

We have implemented a persistence example which showcases how to persist builtin objects without application-visible intrusive changes. It gives complete and orthogonal control over storage and policy issues. One may run this building block example both with PyPy on top of CPython or on translated PyPy versions that include the transparent proxy option.

For more efficient storage, one could easily extend the example to only record changes to an object instead of triggering a full pickle each time. The important point, however, is that an application has direct control on what gets persisted and policies only need to get encoded in one place - the rest of the application can stay unaware of the fact that it is dealing with persistent objects. This means that persistence of data values can also be introduced after an application has been written. Certainly, there can be issues with the reachability graph - objects may reference other objects which should not take part in persistence - however, determining application-specific barriers for pickling the graph of objects is an easier task than pruning the graph of all reachable objects of a running computation.

After further refinements of the transparent proxy mechanism, it will be worthwhile to design practical mechanisms for controlling and implementing effective orthogonal persistence for application objects. Such mechanisms will be able to make use of existing persistence solutions ((ZODB), (PyPersyst)), as the implementation of storage is an orthogonal issue to the question of implementing control and interception mechanisms on individual language level operations. Our prototype implementation of Transparent proxies shows that PyPy's interpreter architecture can easily provide such control irrespective of choice of backend or other variations and extensions to the language, effectively implementing all of the principles of Orthogonal Persistence.

http://www.ecma-international.org/publications/standards/Ecma-335.htm.

http://www.ecma-international.org/publications/standards/Ecma-262.htm

5 Glossary of Abbreviations

The following abbreviations may be used within this document:

5.1 Technical Abbreviations:

AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple Direct-Media Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.

pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

5.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

(OP) Atkinson, M.P. and Morrison R. *Orthogonal Persistent Object Systems*, VLDB Journal, 4(3), pp319-401, 1995, http://citeseer.ist.psu.edu/atkinson95orthogonally.html

(OPJ) Jordan, M. and Atkinson, M. *Orthogonal Persistence for the Java Platform - Draft Specification*, Sun Microsystems Laboratories, October 1999, http://research.sun.com/forest/COM.Sun.Labs.Forest.doc.opjspec.paper_pdf.pdf

(VMC) Rigo, A. and Pedroni, S. *PyPy's approach to virtual machine construction*, in OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, pp. 944-953, ACM Press, 2006

(RC) Miller M.S., *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*, PhD thesis, Johns Hopkins University, Baltimore, 2006

(JFLOW) Myers, A. C. 1999. *JFlow: practical mostly-static information flow control.* In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Antonio, Texas, United States, January 20 - 22, 1999). POPL '99. ACM Press, New York, NY, 228-241.

(MLS) M. Franz. *Moving Trust Out of Application Programs: A Software Architecture Based on Multi-Level Security Virtual Machines*, Technical Report No. 06-10, Donald Bren School of Information and Computer Science, University of California, Irvine; August 2006

(REMOTING) Obermeyer, P. and Hawkins, J. Microsoft .NET Remoting: A Technical Overview. MSDN, 2001. http://msdn2.microsoft.com/en-us/library/ms973857.aspx

(TP) Brumme C., TransparentProxy. Personal Weblog. July 2003. http://blogs.msdn.com/cbrumme/archive/2003/07/14/51495.aspx

(SPREAD) Twisted Spread, Twisted project website, http://twistedmatrix.com/projects/core/spread

(PYRO) Python Remote Objects, project website, http://pyro.sourceforge.net/

(KAWA) Per Bothner, *Kawa -- compiling dynamic languages to the Java VM*, In Proceedings of the USENIX 1998 Technical Conference, FREENIX Track, New Orleans, LA, June 1998. USENIX.

(JYTHON) Jim Hugunin, *Python and Java: The Best of Both Worlds*, Proceedings of the 6th International Python Conference, San Jose, CA, USA, 1997.

(PYDOTNET) Antonio Cuni, *Implementing Python in .NET*, Master thesis, DISI - Università degli Studi di Genova, 2006.

(IRONPYTHON) *IronPython homepage*. http://www.codeplex.com/IronPython

(JYTHON-WEB) *Jython homepage*. http://www.jython.org

(D04.2) *Complete Python Implementation*, PyPy EU-Report, 2005

(D05.1) *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

(D07.1) *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2006

(D13.1) *Build- and Configuration Tool*, PyPy EU-Report, 2007

(ECMA-335) *Standard ECMA-335: Common Language Infrastructure*, ECMA International, June 2005.

(ECMA-262) *ECMAScript specification*.

(CLR) *Microsoft .NET homepage*. http://www.microsoft.com/net/

(Mono) *Mono Project homepage*. http://www.mono-project.com/

(RICHARDS) *Richards benchmark*. http://www.cl.cam.ac.uk/~mr10/Bench.html

(ZODB) *Zope Object Database*. http://www.zope.org/Documentation/Books/ZDG/current/Persistence.stx

(PyPersyst) "Lightweight Data Storage": http://pypersyst.org/

(Jasmin) "Java Assembler": http://jasmin.sourceforge.net/

(PYTHONTYPES) Guido van Rossum, *Unifying types and classes in Python 2.2*, http://www.python.org/download/releases/2.2.3/descrintro/

(PEP252) Guido van Rossum, *Making Types Look More Like Classes*, http://www.python.org/dev/peps/pep-0252/

(PEP253) Guido van Rossum, *Subtyping Built-in Types*, http://www.python.org/dev/peps/pep-0253/

The Standard Object Space

Introduction

The Standard Object Space (StdObjSpace) is the direct equivalent of CPython's object library (the "Objects/" subdirectory in the distribution). It is an implementation of the common Python types in a lower-level language.

The Standard Object Space defines an abstract parent class, W_Object, and a bunch of subclasses like W_IntObject, W_ListObject, and so on. A wrapped object (a "black box" for the bytecode interpreter main loop) is thus an instance of one of these classes. When the main loop invokes an operation, say the addition, between two wrapped objects w1 and w2, the Standard Object Space does some internal dispatching (similar to "Object/abstract.c" in CPython) and invokes a method of the proper W_XyzObject class that can do the operation. The operation itself is done with the primitives allowed by RPython. The result is constructed as a wrapped object again. For example, compare the following implementation of integer addition with the function "int_add()" in "Object/intobject.c":

```
def add__Int_Int(space, w_int1, w_int2):
    x = w_int1.intval
    y = w_int2.intval
    try:
        z = ovfcheck(x + y)
    except OverflowError:
        raise FailedToImplement(space.w_OverflowError,
                                space.wrap("integer addition"))
    return W_IntObject(space, z)
```

Why such a burden just for integer objects? Why did we have to wrap them into W_IntObject instances? For them it seems it would have been sufficient just to use plain Python integers. But this argumentation fails just like it fails for more complex kind of objects. Wrapping them just like everything else is the cleanest solution. You could introduce case testing wherever you use a wrapped object, to know if it is a plain integer or an instance of (a subclass of) W_Object. But that makes the whole program more complicated. The equivalent in CPython would be to use PyObject* pointers all around except when the object is an integer (after all, integers are directly available in C too). You could represent small integers as odd-valuated pointers. But it puts extra burden on the whole C code, so the CPython team avoided it. (In our case it is an optimization that we eventually made, but not hard-coded at this level - see object optimizations.)

So in summary: wrapping integers as instances is the simple path, while using plain integers instead is the complex path, not the other way around.

Object types

The larger part of the StdObjSpace package defines and implements the library of Python's standard built-in object types. Each type (int, float, list, tuple, str, type, etc.) is typically implemented by two modules:

- the *type specification* module, which for a type `xxx` is called `xxxtype.py`;
- the *implementation* module, called `xxxobject.py`.

The `xxxtype.py` module basically defines the type object itself. For example, listtype.py contains the specification of the object you get when you type `list` in a PyPy prompt. listtype.py enumerates the methods specific to lists, like `append()`.

A particular method implemented by all types is the `__new__()` special method, which in Python's new-style-classes world is responsible for creating an instance of the type. In PyPy, `__new__()` locates and imports the module implementing *instances* of the type, and creates such an instance based on the arguments the user supplied to the constructor. For example, tupletype.py defines `__new__()` to import the class `W_TupleObject` from tupleobject.py and instantiate it. The tupleobject.py then contains a "real" implementation of tuples: the way the data is stored in the `W_TupleObject` class, how the operations work, etc.

The goal of the above module layout is to cleanly separate the Python type object, visible to the user, and the actual implementation of its instances. It is possible to provide *several* implementations of the instances of the same Python type, by writing several `W_XxxObject` classes. Every place that instantiates a new object of that Python type can decide which `W_XxxObject` class to instantiate. For example, the regular string implementation is `W_StringObject`, but we also have a `W_StringSliceObject` class whose instances contain a string, a start index, and a stop index; it is used as the result of a string slicing operation to avoid the copy of all the characters in the slice into a new buffer.

From the user's point of view, the multiple internal `W_XxxObject` classes are not visible: they are still all instances of exactly the same Python type. PyPy knows that (e.g.) the application-level type of its interpreter-level `W_StringObject` instances is str because there is a `typedef` class attribute in `W_StringObject` which points back to the string type specification from stringtype.py; all other implementations of strings use the same `typedef` from stringtype.py.

For other examples of multiple implementations of the same Python type, see the object optimizations page.

Multimethods

The Standard Object Space allows multiple object implementations per Python type - this is based on multimethods, although the more precise picture spans several levels in order to emulate the exact Python semantics.

Consider the example of the `space.getitem(w_a, w_b)` operation, corresponding to the application-level syntax `a[b]`. The Standard Object Space contains a corresponding `getitem` multimethod and a family of functions that implement the multimethod for various combination of argument classes - more precisely, for various combinations of the *interpreter-level* classes of the arguments. Here are some examples of functions implementing the `getitem` multimethod:

- `getitem__Tuple_ANY`: called when the first argument is a W_TupleObject, this function converts its second argument to an integer and performs tuple indexing.
- `getitem__Tuple_Slice`: called when the first argument is a W_TupleObject and the second argument is a W_SliceObject. This version takes precedence over the previous one if the indexing is done with a slice object, and performs tuple slicing instead.
- `getitem__String_Slice`: called when the first argument is a W_StringObject and the second argument is a slice object. When the special string slices optimization is enabled, this returns an instance of W_StringSliceObject.
- `getitem__StringSlice_ANY`: called when the first argument is a W_StringSliceObject. This implementation adds the provided index to the original start of the slice stored in the W_StringSliceObject instance. This allows constructs like `a = s[10:100]; print a[5]` to return the 15th character of `s` without having to perform any buffer copying.

Note how the multimethod dispatch logic helps writing new object implementations without having to insert hooks into existing code. Note first how we could have defined a regular method-based API that new object implementations must provide, and call these methods from the space operations. The problem with this approach is that some Python operators are naturally binary or N-ary. Consider for example the addition operation: for the basic string implementation it is a simple concatenation-by-copy, but it can have a rather more subtle implementation for strings done as ropes. It is also likely that concatenating a basic string with a rope string could have its own dedicated implementation - and yet another implementation for a rope string with a basic string. With multimethods, we can have an orthogonally-defined implementation for each combination.

The multimethods mechanism also supports delegate functions, which are converters between two object implementations. The dispatch logic knows how to insert calls to delegates if it encounters combinations of interp-level classes which is not directly implemented. For example, we have no specific implementation for the concatenation of a basic string and a StringSlice object; when the user adds two such strings, then the StringSlice object is converted to a basic string (that is, a temporarily copy is built), and the concatenation is performed on the resulting pair of basic strings. This is similar to the C++ method overloading resolution mechanism (but occurs at runtime).

Multimethod slicing

The complete picture is more complicated because the Python object model is based on *descriptors*: the types `int`, `str`, etc. must have methods `__add__`, `__mul__`, etc. that take two arguments including the `self`. These methods must perform the operation or return `NotImplemented` if the second argument is not of a type that it doesn't know how to handle.

The Standard Object Space creates these methods by *slicing* the multimethod tables. Each method is automatically generated from a subset of the registered implementations of the corresponding multimethod. This slicing is performed on the first argument, in order to keep only the implementations whose first argument's interpreter-level class matches the declared Python-level type.

For example, in a baseline PyPy, `int.__add__` is just calling the function `add__Int_Int`, which is the only registered implementation for `add` whose first argument is an implementation of the `int` Python type. On the other hand, if we enable integers implemented as tagged pointers, then there is another matching implementation: `add__SmallInt_SmallInt`. In this case, the Python-level method `int.__add__` is implemented by trying to dispatch between these two functions based on the interp-level type of the two arguments.

Similarly, the reverse methods (`__radd__` and others) are obtained by slicing the multimethod tables to keep only the functions whose *second* argument has the correct Python-level type.

Slicing is actually a good way to reproduce the details of the object model as seen in CPython: slicing is attempted for every Python types for every multimethod, but the `__xyz__` Python methods are only put into the Python type when the resulting slices are not empty. This is how our `int` type has no `__getitem__` method, for example. Additionally, slicing ensures that `5 .__add__(6L)` correctly returns `NotImplemented` (because this particular slice does not include `add__Long_Long` and there is no `add__Int_Long`), which leads to `6L.__radd__(5)` being called, as in CPython.