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

# Release an RPython Extension Compiler and a Guide Describing the Porting of C Extensions Including Compatibility Considerations

Due date of deliverable: March 2007

Actual Submission date: XXX insert submission date here I

Start date of Project: 1st December 2005 Duration: 2 years

Lead Contractor of this WP: Tismerysoft XXX but seems to be HHU actually

Authors: Armin Rigo (HHU)

Revision: Draft

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 |
|---|---|---|
| 2006-06-16 | Armin Rigo, Gerald Klix | Initial Extension Compiler documentation on-line |
| 2006-12-12 | Armin Rigo | Report on task 1 |
| 2007-01-19 | Armin Rigo | Work on the 2nd RCtypes implementation |
| 2007-01-22 | Armin Rigo | Publishing of intermediate version on the web |

## Abstract

This intermediate report describes the work and results achieved so far in WP03: Synchronization with Standard Python. In particular, we present the RPython Extension Compiler, which allows programmers to write extension modules for multiple Python implementations as a single RPython source. So far, PyPy's translation toolchain can compile such modules either together with the rest of the PyPy interpreter (in which the module ends up as a new built-in module) or as C code that conforms to the C API defined by CPython (producing in that case a dynamically-loaded extension in the CPython style).

## Contents

## 1 Executive Summary

In the field of Python implementations, PyPy is a direct "competitor" to CPython, the existing well-established Python interpreter written in C. One of the main conditions for PyPy to gain acceptance in the community of Python programmers is to keep close to CPython in term of supported features. These features can be naturally divided in two categories: core language features - which are updated at a relatively slow pace by the designers of Python and implemented by them in CPython - and extension modules. The latter are third-party extensions to the CPython interpreter which add new functionality, visible to the programmer as extra modules performing tasks that are not possible or easy to achieve in pure Python code. (A third category of essential feature for Python programmers is the large amount of readily available pure Python libraries, but these should work out-of-the-box on any compliant Python implementation.)

The present report thus covers the tasks of keeping in sync with CPython on the following two fronts:

- Core language features:

  Task 1 of WP03 set out to follow the core CPython language evolution. The Python language went from 2.3 to 2.4 to 2.5 in the past two years. We chose CPython 2.4.1 as our target language. We incorporated a few syntactic features from 2.5, although the latter is a very recent release (September 2006) and not yet widely relied upon.

- Extension modules:

  PyPy's tool-chain is able to translate a subset of the Python language, RPython (D05.1), to other (typically lower-level) languages and platforms. PyPy's own extension modules are written in RPython and based on the Object Space API, core to the PyPy interpreter's approach (D04.2). The long-term approach that we selected and implemented to ease synchronization of the multiple Python implementations is to make it convenient for developers of future CPython modules to write such modules in RPython as well. For this purpose, we offer a stand-alone tool, the Extension Compiler, based on our translation toolchain and able to compile RPython Extension Modules to CPython Extension Modules. The same RPython Extension Modules can be compiled within PyPy.

## 2 Introduction

### 2.1 Purpose of this Document

XXX

### 2.2 Scope of this Document

XXX

### 2.3 Related Documents

XXX

## 3 Core language features

### 3.1 Python releases

During the past two years, the following CPython versions were released:

- Python 2.4 (November 2004)
- Python 2.3.5 (February 2005)
- Python 2.4.1 (March 2005)
- Python 2.4.2 (September 2005)
- Python 2.4.3 (March 2006)
- Python 2.5 (September 2006)
- Python 2.4.4 (October 2006)
- Python 2.3.6 (October 2006)

Last-digit-revisions correspond to bug fix releases. It is customary for a Python version to become widely accepted and relied upon after the ".1" or ".2" release has been made. When PyPy was started, we followed the most recent version at the time, which was the 2.3.x branch.

Predicting the above timeline it was clear from the beginning that we would have to switch to the 2.4 branch. The corresponding work was achieved during summer 2005, when we decided that PyPy should target for the foreseeable future compatibility with Python 2.4.x, and more precisely CPython 2.4.1.

The highlights of this work precisely correspond to the document *What's New in Python 2.4* (PY24), included in annex XXX.

In prevision to the Python 2.5 release we incorporated a few of the syntactic extensions that the Python 2.5 language defines, namely Conditional Expressions (PEP308) and the `with` statement (PEP343). Some other 2.5 language changes are clean-ups of historical accidents of the Python development and have always been available in PyPy, e.g. Exceptions as New-Style Classes (PEP352) and container sizes not limited to 32-bit numbers on 64-bit machines ("Using ssize_t as the index type" (PEP353)). Finally, some features are yet to be implemented, e.g. the new `__index__` method (PEP357).

It should be noted that at least for the new features that we implemented so far, the expressivity of writing them in Python instead of C and the flexibility of relevant parts of the design (e.g. our bytecode compiler) made the task fairly easy.

Nevertheless, we decided to not switch our primary target from Python 2.4.1 to Python 2.5 yet for a number of reasons, in addition to the practical reason that 2.5 was only released late during PyPy's project (near the start of our last phase). The main reason is that Python 2.5 is not yet widely deployed, so that programmers still restrain from using the new features for the time being. As occurred with 2.4, it is likely that PyPy will switch to the 2.5.x branch when Python reaches the maturity of 2.5.1 or 2.5.2.

In summary, the main technical work that classifies under WP03 Task 1 has been the switch from targetting the version 2.3 to the version 2.4 of the language, which occurred soon after the start of the project's funding. More recently, we integrated some features from the version 2.5 but on a minor scale, chosing instead to keep version 2.4 as our primary target for the time being.

## 3.2 CPython community collaboration

A high proportion of the PyPy developers turned out to be regular CPython contributors and community members. Collaboration between PyPy and the rest of the community was successfully carried on by these persons, both on a personal level and via mailing list interactions. To reach the larger community, we presented multiple aspects of PyPy in one or several talks at various conferences, including the major Python conferences. Technical talks have been given at the following events:

- 21C3 2004
- 22C3 2005
- LaTe Lunch 2006
- ACCU 2006
- DLS 2006 (OOPSLA)
- EuroPython 2004
- EuroPython 2005
- EuroPython 2006
- OSCON 2003
- PyCon 2004
- PyCon 2005
- PyCon 2006
- Vancouver Python Workshop 2006

Note that 2003 and 2004 conferences are not part of the present EU project; we mention them because the talks were given by people now part of the EU project. Not listed above, though, are talks from people outside the EU project that talked about PyPy at conferences that we could not attend, e.g. OSDC 2006 (Australia). Most of these talks have triggered interesting collaborations between EU project members and non-EU project members about PyPy.

XXX Check if this list is complete and expand the titles if necessary. XXX Check if this list is *needed* here.

On another level: for the purpose of tracking the changes occurring in CPython, any relevant change - both at the language specification level and at the level of "semi-internal" details that users might nevertheless rely upon - is unlikely to get simply overlooked. Such changes are either well-documented or well known to the members of the PyPy projects that are also active CPython contributors. For this reason we judged superfluous to develop ways to automate the process of tracking CPython developments, beyond the usual tools, i.e. CPython's mailing list, bug and patch tracker, and revision history.

# 4 Extension modules

In regular Python, the ability for users to write external modules is of great importance. These external modules must be written in C, which is both an advantage - it allows access to external C libraries, low-level features, or raw performance - and an inconvenience. In the context of PyPy the greatest drawback of hard-coded C external modules is that low-level details like multithreading (e.g. locks) and memory management must be explicitly written in such a language, which would prevent the same module from being used in several different versions of pypy-c.

## 4.1 The Extension Compiler

### 4.1.1 Introduction

PyPy provides instead a generic way to write modules for all Python implementations: the so-called *mixed module* approach. A single mixed module is implemented as a set of Python source files, making a subpackage of the *pypy/module/* package. While running PyPy, each of these subpackages appears to be a single module, whose interface is specified in the `__init__.py` of the subpackage, and whose implementation is lazily loaded from the various other `.py` files of the subpackage. This was done and documented in (D04.2) already.

The subpackage is written in a way that allows it to be reused for non-PyPy implementations of Python. The goal of the *Extension Compiler* is to compile a mixed module into an extension module for these other Python implementation (so far, this means CPython only).

This means that you can do the following with a mixed module:

- run it on top of CPython. This uses the CPy Object Space to directly run all space operations. Example:

```
$ python
>>> from pypy.interpreter.mixedmodule import testmodule
>>> demo = testmodule("_demo")
[ignore output here]
>>> demo.measuretime(1000000, long)
5
```

- compile it as a CPython extension module. Example:

```
$ python pypy/bin/compilemodule.py _demo
[lots of output]
Created '/tmp/usession-5/_demo/_demo.so'.
$ cd /tmp/usession-5/_demo
$ python
>>> import _demo
>>> _demo.measuretime(10000000, long)
2
```

- run it with PyPy on top of CPython. Example:

```
$ python pypy/bin/py.py --usemodules=_demo
PyPy 0.9.0 in StdObjSpace on top of Python 2.4.4c0 (startuptime: 2.47 secs)
>>>> import _demo
```

```
>>>> _demo.measuretime(10000, long)
[ignore output here]
4
```

- compile it together with PyPy. It becomes a built-in module of `pypy-c`. Example:

```
$ cd pypy/translator/goal
$ python translate.py targetpypystandalone --usemodules=_demo
[wait for 1/2 to 1 hour]
[translation:info] created: ./pypy-c
(Pbd)
$ ./pypy-c
Python 2.4.1 (pypy 0.9.0 build 28xxx) on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>> import _demo
>>>> _demo.measuretime(10000000, long)
2
```

#### 4.1.2 Using the Extension Compiler

New modules go into their own subdirectory of *pypy/module/*. See the following directories as guidelines:

- *pypy/module/_demo*

  A demo module showing a few of the more interesting features.

- *pypy/module/readline*

  A tiny, in-progress example giving bindings to the GNU `readline` library.

- *pypy/module/_sre*

  An algorithmic example: the regular expression engine of CPython, rewritten in RPython.

Modules can be based on ctypes. This is the case in the *pypy/module/readline* and *pypy/module/_demo* examples: they use ctypes to access functions in external C libraries. When translated to C, the calls in these examples become static, regular C function calls -- which means that most of the efficiency overhead of using ctypes disappears during the translation. However, some rules must be followed in order to make ctypes translatable; more information is provided in the section RCtypes below.

All these modules are called "mixed" because they mix interpreter-level and application-level submodules to present a single, coherent module to the user.

More details about how to implement, test and translate a mixed modules for both PyPy and CPython are given in our on-line documentation. They are beyond the scope of the present report. The interested reader is referred to http://codespeak.net/pypy/dist/pypy/doc/extcompiler.html, which is an extended version of the present chapter.

## 4.2 RCtypes

ctypes (CTYPES) is a library for CPython that has been around for some years, and that is now integrated in Python 2.5. It allows Python programs to directly perform calls to external libraries written in C, by declaring function prototypes and all necessary data structures dynamically in Python. It is based on the platform-dependent libffi library (LIBFFI).

*RCtypes* is ctypes restricted to be used from RPython programs. Basically, an RPython program that is going to be translated to C or a C-like language can use (a subset of) ctypes more or less normally.

RCtypes is particularly useful in conjunction with mixed modules and the Extension Compiler: it lets the RPython part of mixed modules use C resources directly, just like ctypes let normal Python programs use C resources. The difference, however, is that during the translation of the RPython module or program to C, the source code -- which contains dynamic foreign function calls written using ctypes -- becomes static C code that performs the same calls. In other words, once compiled within an RPython program, ctypes constructs are all replaced by regular, static C code that no longer uses the *libffi* library. This makes ctypes a very good way to interface with any C library -- not only from plain Python, but from any program or extension module that is written in RPython.

This document assumes some basic familiarity with ctypes. It is recommended to first look up the ctypes documentation at (CTYPES). More details about both ctypes and RCtypes are also available in our on-line documentation at http://codespeak.net/pypy/dist/pypy/doc/rctypes.html.

### 4.2.1 Restrictions

The main restriction that RCtypes adds over ctypes is similar to the general restriction that RPython adds over Python: you cannot be "too dynamic" at run-time, but you can be as dynamic as you want during bootstrapping, i.e. when the source code is imported.

For RCtypes, this means that all ctypes type and function declarations must be done while bootstrapping. You cannot declare new types and functions from inside RPython functions that are going to be translated to C. In practice, this is a fairly soft limitation: most modules using ctypes would typically declare all functions globally at module-level anyway.

Apart from this, many ctypes constructs can be freely used: all primitive types except c_float and c_wchar and c_wchar_p (they will be added at a later time), pointers, structures, arrays, external functions. The following ctypes functions are supported at run-time: create_string_buffer(), pointer(), POINTER(), sizeof(), cast(). (Adding support for more is easy.) There is some support for Unions, and for callbacks and a few more obscure ctypes features, but e.g. custom allocators and return-value-error-checkers are not supported.

## 4.3 RCtypes implementation design

We have experimented with two different implementation approaches. The first one is relatively mature, but does not do the right thing about memory management issues in the most advanced cases. Moreover, it prevents the RPython program to be compiled with a Moving Garbage Collector, which has been the major factor stopping us from experimenting with advanced GCs so far. It is described in the section RCtypes implemented in the RTyper.

The alternative implementation of RCtypes is work in progress; its approach is described in the section RCtypes implemented via a pure RPython interface.

### 4.3.1 RCtypes implemented in the RTyper

The currently available implementation of RCtypes works by integrating itself within the annotation and most importantly RTyping process of the translation toolchain (D05.1): the ctypes objects that the RPython program uses, and the operations it performs on them, are individually replaced during RTyping by sequences of low-level operations performing the equivalent operation in a C-like language.

### 4.3.2 Annotation

Ctypes on CPython tracks all the memory it has allocated by itself, may it be referenced by pointers and structures or only pointers. Thus memory allocated by ctypes is properly garbage collected and no dangling pointers should arise.

Pointers to structures returned by an external function are a different story. For such pointers we have to assume that they were not allocated by ctypes, and just keep a pointer to the memory, hoping that it will not go away. This is the same as in ctypes.

For objects that are known to be allocated by RCtypes, a simple GC-aware box suffices to hold the needed memory. For other objects, an extra indirection is required. Thus we use the annotator to track which objects are statically known to be allocated by RCtypes.

### 4.3.3 RTyping: Memory layout

In ctypes, all instances are mutable boxes containing either some raw memory with a layout compatible to that of the equivalent C type, or a reference to such memory. The reference indirection is transparent to the user; for example, dereferencing a ctypes object "pointer to structure" results in a "structure" object that doesn't include a copy of the data, but only a reference to that data. (This is similar to the C++ notion of reference: it is just a pointer at the machine level, but at the language level it behaves like the object that it points to, not like a pointer.)

The rtyper maps this to the LLType model as follows. For boxes that embed the raw memory content (i.e. are known to own their memory):

```
Ptr( GcStruct( "name_owner",
        ("c_data", Struct(...) ) ) )
```

where the raw memory content and layout is specified by the "Struct(...)" part.

The key point here is that the "Struct(...)" part follows exactly the C layout, with no extra headers. If it contains pointers, they point to exact C data as well, without headers, and so on. From C's point of view this is all perfectly legal C data in the format expected by the external C library -- because the C library never sees pointers to the whole GcStruct boxes.

In other words, we have a two-level picture of how data looks like in a translated RPython program: at the RPython level, it contains a graph of GcStructs, with GC headers, and sometimes fields pointing at each other for the purpose of keeping the necessary GcStructs alive. At the same time, *inside* these GcStruct, there is a part that looks exactly like C data structures. This part can contain pointers to other C data structures, which may live within other GcStructs. So the C libraries will only see and modify the C data that is embedded in the "c_data" substructure of GcStructs. This is somewhat similar to the idea that if you call malloc() in a C program, you actually get a block of memory that is embedded in some larger block with malloc-specific headers; but at the level of C it doesn't matter because you only pass around and store pointers to the "embedded sub-blocks" of memory, never to the whole block with all its headers.

The detail of the memory layout for each type of ctypes object can be looked up in our on-line documentation at http://codespeak.net/pypy/dist/pypy/doc/rctypes.html#rtyping-memory-layout.

### 4.3.4 RCtypes implemented via a pure RPython interface

As of January 2007, a second implementation of RCtypes is being developed. The basic conclusion on the work on the first implementation is that it is quite tedious to have to write code that generates detailed low-level operations for each high-level ctypes operation - and most importantly, it is not flexible enough for our purposes. The major issue is that problems were recently found in the corner cases of memory management, i.e. when ctypes objects should keep other ctypes object alive and for how long. Fixing this would involve some pervasive changes in the low-level representation of ctypes objects, which would require all the RCtypes RTyping code to be updated. Similarly, to be able to use a Moving Garbage Collector, the memory that is visible to the external C code need to be separated from the GC-managed memory that is used for the RCtypes objects themselves; this would also require a complete upgrade of all the RTyping code.

### 4.3.5 The rctypesobject RPython library

To solve this, we started by writing a complete RPython library that offers the same functionality as ctypes, but with a more regular, RPython-compatible interface. This library is in the *pypy/rlib/rctypes/rctypesobject.py* module. All operations use either static or instance methods on classes following a structure similar to the ctypes classes. The flexibility of writing a normal RPython library allowed us to use internal support classes freely, together with data structures appropriate to the memory management needs.

This interface is more tedious to use than ctypes' native interface, but it not meant for manual use. Instead, the translation toolchain maps uses of the usual ctypes interface in the RPython program to the more verbose but more regular rctypesobject interface.

The rctypesobject library is mostly implemented as a family of functions that build and return new classes. For example, `Primitive(Signed)` return the class corresponding to C objects of the low-level type `Signed` (corresponding to `long` in C), `RPointer(Primitive(Signed))` is the class corresponding to the type "pointer to long" and `RVarArray(RPointer(Primitive(Signed)))` return the class corresponding to arrays of pointers to longs.

Each of these classes expose a similar interface: the `allocate()` static method returns a new instance with its own memory storage (allocated with a separate `lltype.malloc()` as regular C memory, and automatically freed from the `__del__()` method of the instance). For example, `Primitive(Signed).allocate()` returns an instance that manages a word of C memory, large enough to contain just one `long`. If `x` designates this instance, the expression `pointer(x)` allocates and returns an instance of class `RPointer(Primitive(Signed))` that manages a word of C memory, large enough to hold a pointer, and initialized to point to the previous `long` in memory.

The details of the interface can be found in the library's accompanying test file, *pypy/rlib/rctypes/test/test_rcty*

#### 4.3.6 ControllerEntry

We implemented a generic way in the translation toolchain to map operations on arbitrary Python objects to calls to "glue" RPython classes and methods. This mechanism can be found in *pypy/rpython/controllerentry.py*. In short, it allows code to register, for each full Python object, class, or metaclass, a corresponding "controller" class. During translation, the controller is invoked when the Python object, class or metaclass is encountered. The annotator delegates to the controller the meaning of all operations - instantiation, attribute reading and setting, etc. By default, this delegation is performed by replacing the operation by a call to a method of the controller; in this way, the controller can be written simply as a family of RPython methods with names like:

- `new()` - called when the source RPython program tries to instantiate the full Python class
- `get_xyz()` and `set_xyz()` - called when the RPython program tries to get or set the attribute `xyz`
- `getitem()` and `setitem()` - called when the RPython program uses the `[ ]` notation to index the full Python object

If necessary, the controller can also choose to entierely override the default annotating and rtyping behavior and insert its own. This is useful for cases where the method cannot be implemented in RPython, e.g. in the presence of a polymorphic operation that would cause the method signature to be ill-typed.

For RCtypes, we implemented controllers that map the regular ctypes objects, classes and metaclasses to classes and operations from rctypesobject. This turned out to be a good way to separate the RCtypes implementation issues from the (sometimes complicated) interpretation of ctypes' rich and irregular interface.

#### 4.3.7 Performance

The greatest advantage of the ControllerEntry approach over the direct RTyping approach of the first RCtypes implementation is its higher level, giving flexibility. This is also potentially a disadvantage: there is for example no owner/alias analysis done during annotation; instead, all ctypes objects of a given type are implemented identically. We think that the general optimizations that we implemented - most importantly malloc removal (D07.1) - are either good enough to remove the overhead in the common case, or can be made good enough with some more efforts.

XXX work in progress.

# 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. |
| 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. |
| Graphviz | Graph visualisation software from AT&T. |
| 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 Directmedia Library - 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

(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

(PY24) *What's New in Python 2.4*, A.M. Kuchling, Python Software Foundation, http://www.python.org/doc/2.4/whatsnew/whatsnew24.html

(PEP308) http://www.python.org/dev/peps/pep-0308/

(PEP343) http://www.python.org/dev/peps/pep-0343/

(PEP352) http://www.python.org/dev/peps/pep-0352/

(PEP353) http://www.python.org/dev/peps/pep-0353/

(PEP357) http://www.python.org/dev/peps/pep-0357/

(CTYPES) *The ctypes package*, Thomas Heller, http://starship.python.net/crew/theller/ctypes/

(LIBFFI) *The libffi Home Page*, Anthony Green, http://sources.redhat.com/libffi/