IST FP6-004779

PYPY

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

STREP

IST Priority 2

D04.2: Complete Python implementation running on top of CPython

Due date of deliverable: August 2005

Actual Submission date: 23th December 2005

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

Lead Contractor of this WP: Strakt

Authors: Jacob Hallén (AB Strakt), Christian Tismer (tismerysoft), the PyPy team (included documentation)

Revision: final

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

Dissemination Level: PU (Public)

Abstract

This document describes the implementation of Python on top of CPython as available in the 0.7 release of the PyPy project, made on Sun 28 Aug 28 2005.

Contents

1 Executive Summary

This release of the PyPy interpreter is built to run on top of the CPython interpreter and it is an extention of the work supplied in deliverable D04.1. Between the releases 0.6 and 0.7 efforts have been concentrated on making the implementation of Python as close to complete as possible. The stated goal of passing more than 90% of the official Python test suite for core language features has been reached. While running on top of CPython, the interpreter is still about 2000 times slower than CPython. This is expected.

2 Introduction

2.1 Purpose of this Document

This document describes the release 0.7 version of the PyPy interpreter. It serves to document when a mostly complete interpreter on top of CPython was released and to what extent it was compatible with CPython. In order to facilitate the understanding of the PyPy project, basic documentation on architecture, the interpreter, object spaces and project coding conventions have been included in this deliverable. Due to an omission in the Description of Work, there was no deliverable for these documents. The documents were primarily prepared as an introduction for people taking an interest in the project and for people who want to join in the coding effort. This may sometimes be reflected in the wording and style.

2.2 Scope of this Document

This document describes basic interpretation of Python code. It does not deal with any aspects of the translation of the PyPy interpreter into lower level code nor any aspects of optimization, even though such elements were part of the 0.7 release of the project.

2.3 Related Documents

This document has been prepared in conjunction with other tasks in work packages 3, 4 and 5. Before reading this document, a close look at the following deliverable is recommended.

- D04.1 First partial Python Implementation on top of CPython

3 Achieved Goals

The second public release of the PyPy interpreter was released on Sun 28 Aug 2005, with version number 0.7.0.

The release is available at

- http://code2.codespeak.net/download/pypy/pypy-0.7.0.tar.bz2,
- http://code2.codespeak.net/download/pypy/pypy-0.7.0.tar.gz and

- http://code2.codespeak.net/download/pypy/pypy-0.7.0.zip.

The release completes task 2 of WP 4:

> *Port the standard Python "builtin" library:*
>
> - *types (there are some 100s of them)*
> - *built-in functions*
> - *built-in modules*
> - *other built-in objects, e.g. exception classes*
>
> *Research and decide on a case-by-case basis how to implement each one within PyPy in a simple and efficient way. The common options are to either re-implement them inside the interpreter where appropriate, or to provide a pure Python replacement (which in some cases already exists and can be borrowed).*

There are still a few features that are unsupported, with *__del__* being the most important one to deal with. The goal of supporting more than 90% of the core modules in the standard library has been accomplished as well. The few remaining core modules are in general fairly difficult to implement. The *weakref* module is essential for many of the non-core modules and should be implemented fairly soon. The challenges it poses have a lot in common with the challenges of *__del__*. The *threading* modules could be omitted in favour of other concurrency mechanisms. A decision on this issue will be made later in the project. The other core modules that do not pass the compliance tests have dependencies on *weakref* or pose challenges which slow down the implementation process. None of the failing modules block the progress of other work packages.

4 Public Announcement of the Release

The following release announcement was sent out to various mailing lists:

```
pypy-0.7.0: first PyPy-generated Python Implementations
+++++++++++++++++++++++++++++++++++++++++++++++++++++++

What was once just an idea between a few people discussing
on some nested mailing list thread and in a pub became reality ...
the PyPy development team is happy to announce its first
public release of a fully translatable self contained Python
implementation.  The 0.7 release showcases the results of our
efforts in the last few months since the 0.6 preview release
which have been partially funded by the European Union:

- whole program type inference on our Python Interpreter
  implementation with full translation to two different
  machine-level targets: C and LLVM

- a translation choice of using a refcounting or Boehm
  garbage collectors

- the ability to translate with or without thread support
- very complete language-level compliancy with CPython 2.4.1


What is PyPy (about)?
+++++++++++++++++++++

PyPy is a MIT-licensed research-oriented reimplementation of
Python written in Python itself, flexible and easy to
experiment with.  It translates itself to lower level
languages.  Our goals are to target a large variety of
platforms, small and large, by providing a compilation toolsuite
that can produce custom Python versions.  Platform, Memory and
Threading models are to become aspects of the translation
process - as opposed to encoding low level details into a
language implementation itself.  Eventually, dynamic
optimization techniques - implemented as another translation
aspect - should become robust against language changes.

Note that PyPy is mainly a research and development project
and does not by itself focus on getting a production-ready
Python implementation although we do hope and expect it to
become a viable contender in that area sometime next year.


Where to start?
+++++++++++++++

Getting started:    http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

PyPy Documentation: http://codespeak.net/pypy/dist/pypy/doc/

PyPy Homepage:      http://codespeak.net/pypy/

The interpreter and object model implementations shipped with
the 0.7 version can run on their own and implement the core
language features of Python as of CPython 2.4.  However, we still
do not recommend using PyPy for anything else than for education,
playing or research purposes.

Ongoing work and near term goals
++++++++++++++++++++++++++++++++

PyPy has been developed during approximately 15 coding sprints
across Europe and the US.  It continues to be a very
dynamically and incrementally evolving project with many
one-week meetings to follow.  You are invited to consider coming to
the next such meeting in Paris mid October 2005 where we intend to
plan and head for an even more intense phase of the project
involving building a JIT-Compiler and enabling unique
features not found in other Python language implementations.

PyPy has been a community effort from the start and it would
not have got that far without the coding and feedback support
from numerous people.   Please feel free to give feedback and
raise questions.

    contact points: http://codespeak.net/pypy/dist/pypy/doc/contact.html
    contributor list: http://codespeak.net/pypy/dist/pypy/doc/contributor.html

have fun,

    the pypy team, of which here is a partial snapshot
    of mainly involved persons:

    Armin Rigo, Samuele Pedroni,
    Holger Krekel, Christian Tismer,
    Carl Friedrich Bolz, Michael Hudson,
    Eric van Riet Paap, Richard Emslie,
    Anders Chrigstroem, Anders Lehmann,
    Ludovic Aubry, Adrien Di Mascio,
    Niklaus Haldimann, Jacob Hallen,
    Bea During, Laura Creighton,
    and many contributors ...

PyPy development and activities happen as an open source project
and with the support of a consortium partially funded by a two
year European Union IST research grant. Here is a list of
the full partners of that consortium:

    Heinrich-Heine University (Germany), AB Strakt (Sweden)
    merlinux GmbH (Germany), tismerysoft GmbH(Germany)
    Logilab Paris (France), DFKI GmbH (Germany)
    ChangeMaker (Sweden), Impara (Germany)
```

5 Compliance Test Results

The following list shows to what extent the 0.7.0 release passed the Python compliance tests. These tests were taken from the 2.4.1 release of CPython and in some cases modified by the PyPy project in order to remove implementation dependent details from the tests. In a few cases standard library modules implemented in Python that are provided in the 2.4.1 release of CPython had to be modified. All modified tests and modified modules are found in the *lib-python/modified-2.4.1/* subdirectory of the release. The complete original standard library tests, included the ones that were necessary to modify, can be found in the *lib-python/2.4.1/* subdirectory of the release. Thus it is very easy to compare directories and files in order to determine exactly which changes have been necessary to make. Please note that between the 0.6.1 and the 0.7.0 releases of PyPy, there was a change from being compliant with Python 2.3.4 to being compliant with Python 2.4.1. This required quite a bit of effort. It also resulted in the compliance tests results in Deliverable D04.1 not being fully comparable to the results in this deliverable.

5.1 Core Tests

Total test compliance	94.65%
Test modules passed completely	89.92%
Test modules (partially) failed	8.40%
Test modules timeout	1.68%

Details

Percent failed	Test
100.00%	test_imp
100.00%	test_weakref
100.00%	test_importhooks
66.67%	test_traceback
13.89%	test_trace
12.34%	test_generators
100.00%	test_threading
100.00%	test_thread
0.00%	test_future1
0.00%	test_augassign
0.00%	test_softspace
0.00%	test_eof
0.00%	test___future__
0.00%	test_future3
0.00%	test_hash
0.00%	test_contains
0.00%	test_unary
0.00%	test_charmapcodec
0.00%	test_opcodes
0.00%	test_types
0.00%	test_decorators
0.00%	test_atexit
0.00%	test_call
0.00%	test_new
0.00%	test_iterlen
0.00%	test_time
0.00%	test_multibytecodec_support
0.00%	test_math
0.00%	test_long_future
0.00%	test_dircache
0.00%	test_cmath
0.00%	test_hexoct
0.00%	test_scope
0.00%	test_syntax
0.00%	test_warnings
0.00%	test_pkg
0.00%	test_univnewlines
0.00%	test_fnmatch
0.00%	test_isinstance
0.00%	test_future2

Percent failed	Test
0.00%	test_threaded_import
0.00%	test_dummy_threading
0.00%	test_StringIO
0.00%	test_filecmp
0.00%	test_binop
0.00%	test_grammar
0.00%	test_profilehooks
0.00%	test_future
0.00%	test_fileinput
0.00%	test_pkgimport
0.00%	test_glob
0.00%	test_operator
0.00%	test_dummy_thread
0.00%	test_string
0.00%	test_bool
0.00%	test_userdict
0.00%	test_copy_reg
0.00%	test_richcmp
0.00%	test_os
0.00%	test_str
0.00%	test_extcall
0.00%	test_compare
0.00%	test_transformer
0.00%	test_builtin
0.00%	test_fpformat
0.00%	test_iter
0.00%	test_global
0.00%	test_longexp
0.00%	test_getopt
0.00%	test_unittest
0.00%	test_import
0.00%	test_pprint
0.00%	test_threading_local
0.00%	test_bisect
0.00%	test_codecs
0.00%	test_marshal
0.00%	test_pow
0.00%	test_heapq
0.00%	test_sort
0.00%	test_pickle
0.00%	test_unicode
0.00%	test_long

Percent failed	Test
0.00%	test_codeccallbacks

5.2 Non-core tests

Total test compliance	42.97%
Test modules passed completely	36.78%
Test modules (partially) failed	61.49%
Test modules timeout	1.72%

Details

Percent failed	Test
100.00%	test_crypt
100.00%	test_frozen
100.00%	test_timing
100.00%	test_symtable
100.00%	test_cd
100.00%	test_sunaudiodev
100.00%	test_gdbm
100.00%	test_dl
100.00%	test_capi
100.00%	test_bsddb185
100.00%	test_al
100.00%	test_resource
100.00%	test_signal
100.00%	test_rgbimg
100.00%	test_fork1
100.00%	test__locale
100.00%	test_bsddb3
100.00%	test_select
100.00%	test_cl
100.00%	test_imgfile
100.00%	test_linuxaudiodev
100.00%	test_winsound
100.00%	test_aepack
100.00%	test_imageop
100.00%	test_grp
100.00%	test_plistlib
100.00%	test_hotshot
100.00%	test_tcl

Percent failed	Test
100.00%	test_getargs
100.00%	test_gl
100.00%	test_winreg
100.00%	test_curses
100.00%	test_audioop
100.00%	test_ioctl
100.00%	test_pyexpat
100.00%	test_locale
100.00%	test_macostools
100.00%	test_applesingle
100.00%	test_dbm
100.00%	test_gc
100.00%	test_zlib
100.00%	test_structseq
100.00%	test_poll
100.00%	test_gzip
100.00%	test_normalization
100.00%	test_bz2
100.00%	test_ossaudiodev
100.00%	test_zipimport
100.00%	test_pep277
100.00%	test_codecmaps_hk
100.00%	test_pep263
100.00%	test_codecmaps_cn
100.00%	test_codecmaps_kr
100.00%	test_codecmaps_tw
100.00%	test_pty
100.00%	test_threadsignals
100.00%	test_socket_ssl
100.00%	test_scriptpackages
100.00%	test_codecmaps_jp
100.00%	test_socketserver
100.00%	test_site
100.00%	test_sax
100.00%	test_pwd
100.00%	test_nis
100.00%	test_mmap
100.00%	test_timeout
100.00%	test_bsddb
100.00%	test_queue
100.00%	test_codecencodings_tw
100.00%	test_regex

Percent failed	Test
100.00%	test_strop
100.00%	test_unicode_file
100.00%	test_codecencodings_hk
100.00%	test_getargs2
100.00%	test_multibytecodec
100.00%	test_macfs
100.00%	test_csv
100.00%	test_socket
100.00%	test_codecencodings_kr
100.00%	test_codecencodings_cn
100.00%	test_subprocess
100.00%	test_random
100.00%	test_codecencodings_jp
100.00%	test_urllibnet
40.00%	test_dis
100.00%	test_peepholer
100.00%	test_minidom
46.15%	test_pep292
100.00%	test_profile
61.54%	test_tarfile
100.00%	test_inspect
20.00%	test_mailbox
11.11%	test_distutils
50.00%	test_xmlrpc
100.00%	test_doctest2
2.38%	test_cfgparser
100.00%	test_compiler
50.00%	test_zipfile
100.00%	test_asynchat
100.00%	test_urllib2net
2.15%	test_pickletools
3.57%	test_gettext
28.18%	test_email
3.85%	test_doctest
1.82%	test_datetime
100.00%	test_pyclbr
2.99%	test_decimal
100.00%	test_unicodedata
100.00%	test_difflib
100.00%	test_logging
0.00%	test_bastion
0.00%	test_fcntl

Percent failed	Test
0.00%	test_openpty
0.00%	test_colorsys
0.00%	test_wave
0.00%	test_largefile
0.00%	test_macpath
0.00%	test_shutil
0.00%	test_binhex
0.00%	test_errno
0.00%	test_ntpath
0.00%	test_urlparse
0.00%	test_stringprep
0.00%	test_pep247
0.00%	test_netrc
0.00%	test_popen
0.00%	test_MimeWriter
0.00%	test_commands
0.00%	test_whichdb
0.00%	test_multifile
0.00%	test_rfc822
0.00%	test_anydbm
0.00%	test_posix
0.00%	test_binascii
0.00%	test_posixpath
0.00%	test_calendar
0.00%	test_imaplib
0.00%	test_robotparser
0.00%	test_htmllib
0.00%	test_mimetypes
0.00%	test_cgi
0.00%	test_md5
0.00%	test_shlex
0.00%	test_mimetools
0.00%	test_email_codecs
0.00%	test_textwrap
0.00%	test_hmac
0.00%	test_xmllib
0.00%	test_sgmllib
0.00%	test_dumbdbm
0.00%	test_httplib
0.00%	test_threadedtempfile
0.00%	test_urllib
0.00%	test_cookie

Percent failed	Test
0.00%	test_tokenize
0.00%	test_strptime
0.00%	test_shelve
0.00%	test_sundry
0.00%	test_xpickle
0.00%	test_mhlib
0.00%	test_ucn
0.00%	test_htmlparser
0.00%	test_cookielib
0.00%	test_sets
0.00%	test_popen2
0.00%	test_strftime

6 PyPy - Architecture Overview

This document gives an overview of the goals and architecture of PyPy. See also getting started for a practical introduction.

6.1 Mission Statement

PyPy is an implementation of the Python programming language written in Python itself, flexible and easy to experiment with. Our long-term goals are to target a large variety of platforms, small and large, by providing a compiler toolsuite that can produce custom Python versions. Platform, memory and threading models are to become aspects of the translation process - as opposed to encoding low level details into the language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation aspect - should become robust against language changes.

6.2 PyPy - an Implementation of Python in Python

It has become a tradition in the development of computer languages to implement each language in itself. This serves many purposes. By doing so, you demonstrate the versatility of the language and its applicability for large projects. Writing compilers and interpreters are among the most complex endeavours in software development.

An important aspect of implementing Python in Python is the high level of abstraction and compactness of the language. This allows an implementation that is, in some respects, easier to understand and play with than the one written in C (referred to throughout the PyPy documentation and source as "CPython").

Another central idea in PyPy is building the implementation in the form of a number of independent modules with clearly defined and well tested API's. This eases reuse and allows experimenting with multiple implementations of specific features.

Later in the project we will introduce optimizations, following the ideas of Psyco that should make PyPy run Python programs faster than CPython, and extensions, following the ideas of Stackless and others, that will increase the expressive power available to Python programmers.

6.3 Higher Level Picture

As you would expect from a project implemented using ideas from the world of Extreme Programming, the architecture of PyPy has evolved over time and continues to evolve. Nevertheless, the high level architecture is clear now. There are two independent basic subsystems: the Standard Interpreter and the Translation Process.

6.3.1 The Standard Interpreter

The *standard interpreter* is the subsystem implementing the Python language. It is divided into two components:

- the bytecode interpreter which is responsible for interpreting code objects and implementing bytecodes,
- the standard object space which implements creating, accessing and modifying application level objects.

Note that the *standard interpreter* can run fine on top of CPython if one is willing to pay the performance penalty for double-interpretation.

6.3.2 The Translation Process

The *translation process* aims at producing a different (low-level) representation of our standard interpreter. The *translation process* is done in four steps:

- producing a *flow graph* representation of the standard interpreter. A combination of the bytecode interpreter and a *flow object space* performs *abstract interpretation* to record the flow of objects and execution throughout a Python program into such a *flow graph*;
- the *annotator* which performs type inference on the flow graph;
- the *typer* which, based on the type annotations, turns the flow graph into one using only low-level, C-like operations;
- the *code generator* which translates the resulting flow graph into another language, currently C or LLVM.

See below for the translation process in more details.

6.4 The Bytecode Interpreter

The *plain bytecode interpreter* handles Python code objects. The interpreter can build code objects from Python sources, if needed, by invoking a bytecode compiler. Code objects are a nicely preprocessed, structured representation of source code, and their main content is *bytecode*. We use the same compact bytecode format as CPython 2.4.

Our bytecode compiler is implemented as a chain of flexible passes (tokenizer, lexer, parser, abstract syntax tree builder, bytecode generator). The latter passes are based on the `compiler` package from the standard library of CPython with various improvements and bug fixes. The bytecode compiler (living under *interpreter/astcompiler/*) is now integrated and is translated with the rest of PyPy.

In addition to storing bytecode, code objects also know how to create a *frame* object which is responsible for *interpreting* a code object's bytecode. Each bytecode is implemented by a Python function, which in turn delegates operations on application-level objects to an object space. This interpretation and delegation is the core of the bytecode interpreter.

This part is implemented in the *interpreter/* directory. People who are familiar with the CPython implementation of the above concepts will easily recognize them there. The major differences are the overall usage of the Object Space indirection to perform operations on objects, and the organization of the built-in modules (described here).

6.5 The Object Space

The object space creates all objects and knows how to perform operations on the objects. You may think of an object space as being a library offering a fixed API, a set of *operations* with implementations that correspond to the known semantics of Python objects. An example of an operation is *add*: add's implementations are for example responsible for performing numeric addition when add works on numbers, and concatenation when add works on built-in sequences.

All object-space operations take and return application-level objects. There are only a few very simple object-space operations which allow the bytecode interpreter to gain some knowledge about the value of an application-level object. The most important one is `is_true()`, which returns a boolean interpreter-level value. This is necessary to implement, for example, if-statements (or rather, to be pedantic, to implement the conditional-branching bytecodes into which if-statements get compiled).

We currently have four working object spaces which can be plugged into the bytecode interpreter:

- The *Standard Object Space* is a complete implementation of the various built-in types and objects of Python. The Standard Object Space together with the bytecode interpreter is the foundation of our Python implementation. Internally it is a set of interpreter-level classes implementing the various application-level objects -- integers, strings, lists, types, etc. To draw a comparison with CPython, the Standard Object Space provides the equivalent of the C structures `PyIntObject`, `PyListObject`, etc.
- the *Trace Object Space* wraps e.g. the standard object space in order to trace the execution of bytecodes, frames and object space operations.
- the *Thunk Object Space* wraps another object space (e.g. the standard one) and adds two capabilities: lazily computed objects (computed only when an operation is performed on them), and "become", which completely and globally replaces one object with another.

- the *Flow Object Space* transforms a Python program into a flow-graph representation, by recording all operations that the bytecode interpreter would like to perform when it is shown the given Python program. This technique is explained later in this document.

For a description of the object spaces, please see the *objspace document*. The sources of PyPy contain the various object spaces in the directory *objspace/*.

6.6 Application-level and Interpreter-level Execution and Objects

Since Python is used for implementing all of our code base, there is a crucial distinction to be aware of: that between *interpreter-level* objects and *application-level* objects. The latter are the ones that you deal with when you write normal Python programs. Interpreter-level code, however, cannot invoke operations nor access attributes from application-level objects. You will immediately recognize any interpreter level code in PyPy, because half the variable and object names start with a `w_`, which indicates that they are wrapped application-level values.

Let's show the difference with a simple example. To sum the contents of two variables a and `b`, one would write the simple application-level `a+b` -- in contrast, the equivalent interpreter-level code is `space.add(w_a, w_b)`, where `space` is an instance of an object space, and `w_a` and `w_b` are typical names for the wrapped versions of the two variables.

It helps to remember how CPython deals with the same issue: interpreter level code, in CPython, is written in C and thus code for the addition is typically `PyNumber_Add(p_a, p_b)` where `p_a` and `p_b` are C variables of type `PyObject*`. This is conceptually similar to how we write our interpreter-level code in Python.

Moreover, in PyPy we have to make a sharp distinction between interpreter- and application-level *exceptions*: application exceptions are always contained inside an instance of the class `OperationError`. This makes it easy to distinguish failures (or bugs) in our interpreter-level code from failures appearing in a Python application level program that we are interpreting.

6.6.1 Application Level is Often Preferable

Application-level code is substantially higher-level, and therefore correspondingly easier to write and debug. For example, suppose we want to implement the `update` method of dict objects. Programming at application level, we can write an obvious, simple implementation, one that looks like an **executable definition** of `update`, for example:

```
def update(self, other):
    for k in other.keys():
        self[k] = other[k]
```

If we had to code at interpreter level only, we would have to code something much lower-level and involved, say something like:

```
def update(space, w_self, w_other):
    w_keys = space.call_method(w_other, 'keys')
    w_iter = space.iter(w_keys)
    while True:
        try:
            w_key = space.next(w_iter)
        except OperationError, e:
            if not e.match(space, space.w_StopIteration):
                raise       # re-raise other app-level exceptions
            break
        w_value = space.getitem(w_other, w_key)
        space.setitem(w_self, w_key, w_value)
```

This interpreter-level implementation looks much more similar to the C source code. It is still more readable than its C counterpart because it doesn't contain memory management details and can use Python's native exception mechanism.

In any case it should be obvious that the application-level implementation is definitely more readable, more elegant and more maintainable than the interpreter-level one.

In fact, in almost all parts of PyPy, you find application level code in the middle of interpreter-level code. Apart from some bootstrapping problems (application level functions need a certain initialization level of the object space before they can be executed), application level code is usually preferable. We have an abstraction (called the 'Gateway') which allows the caller of a function to remain ignorant of whether a particular function is implemented at application or interpreter level.

6.6.2 Wrapping

The `w_` prefixes so lavishly used in the previous example indicate by PyPy coding convention that we are dealing with *wrapped* (or *boxed*) objects, that is, interpreter-level objects which the object space constructs to implement corresponding application-level objects. Each object space supplies `wrap`, `unwrap`, `int_w`, `interpclass_w`, etc. operations that move between the two levels for objects of simple built-in types; each object space also implements other Python types with suitable interpreter-level classes with some amount of internal structure.

For example, an application-level Python `list` is implemented by the standard object space as an instance of `W_ListObject`, which has an instance attribute `wrappeditems` (an interpreter-level list which contains the application-level list's items as wrapped objects).

The rules are described in more details in the coding guide.

6.7 RPython, the Flow Object Space and Translation

One of PyPy's now achieved objectives is to enable translation of our bytecode interpreter and standard object space into a lower-level language. In order for our translation and type inference mechanisms to work effectively, we need to restrict the dynamism of our interpreter-level Python code at some point. However, in the start-up phase, we are completely free to use all kinds of powerful Python constructs including metaclasses and execution of dynamically constructed strings. However when the initialization phase (mainly, the function `objspace.initialize()`) finishes, all code objects involved need to adhere to a more static subset of Python: Restricted Python, also known as RPython.

The Flow Object Space then, with the help of our bytecode interpreter, works through those initialized RPython code objects. The result of this abstract interpretation is a flow graph: yet another representation of a Python program, but one which is suitable for applying translation and type inference techniques. The nodes of the graph are basic blocks consisting of Object

Space operations, flowing of values, and an exitswitch to one, two or multiple links which connect each basic block to other basic blocks.

The flow graphs are fed as input into the Annotator. The Annotator, given entry point types, infers the types of values that flow through the program variables. This is the core of the definition of RPython: RPython code is restricted in such a way that the Annotator is able to infer consistent types. How much dynamism we allow in RPython depends on and is restricted by the Flow Object Space and the Annotator implementation. The more we can improve this translation phase the more dynamism we can allow. In some cases, however, it is more feasible and practical to just get rid of some of the dynamism we use in our interpreter level code. It is mainly because of this trade-off situation that the definition of RPython has shifted over time. Although the Annotator is pretty stable now and able to process the whole of PyPy, the Rpython definition will probably continue to shift marginally as we improve it.

The actual low-level code (and, in fact, also other high-level code) is emitted by "visiting" the type-annotated flow graph. Currently we have a C-producing backend and an LLVM-producing backend. The former also accepts non-annotated or partially-annotated graphs, which allows us to test it on a larger class of programs than what the Annotator can (or ever will) fully process.

The newest piece of this puzzle is the *Typer*, which inputs the high-level types inferred by the Annotator and uses them to modify the flow graph in-place to replace its operations with low-level ones, directly manipulating C-like values and data structures.

The complete translation process is described in the translation document in more details. There is a graph that gives an overview of the whole process.

7 PyPy - Bytecode Interpreter

7.1 Introduction and Overview

This document describes the implementation of PyPy's Bytecode Interpreter and related Virtual Machine functionalities.

PyPy's bytecode interpreter has a structure reminiscent of CPython's Virtual Machine: It processes code objects parsed and compiled from Python source code. Code objects contain condensed information about their respective functions, class and module body source codes. Interpreting such code objects means instantiating and initializing a Frame class and then calling its `frame.eval()` method. This main entry point initialises appropriate namespaces and then interprets each bytecode instruction. Python's standard library contains the *lib-python/2.4.1/dis.py* module which allows to view the Virtual's machine bytecode instructions:

```
>>> import dis
>>> def f(x):
        return x + 1
>>> dis.dis(f)
4       0 LOAD_FAST                0 (x)
        3 LOAD_CONST               1 (1)
        6 BINARY_ADD
        7 RETURN_VALUE
```

CPython as well as PyPy are stack-based virtual machines, i.e. they don't have registers but put objects to and pull objects from a stack. The bytecode interpreter is only responsible for implementing control flow and putting and pulling black box objects to and from this value stack. The bytecode interpreter does not know how to perform operations on those black box (wrapped) objects for which it delegates to the object space. In order to implement a conditional branch in a program's execution however, it needs to gain minimal knowledge about a wrapped object. Thus each object space has to offer a `is_true(w_obj)` operation which returns an interpreter-level boolean value.

For the understanding of the interpreter's inner workings it is crucial to recognize the concepts of interpreter-level and application-level code. In short interpreter-level is executed directly on the machine and invoking application-level functions leads to a bytecode interpretation indirection. However, special care must be taken regarding exceptions because application level exceptions are wrapped into `OperationErrors` which are thus distinguished from plain interpreter-level exceptions. See application level exceptions for some more information on `OperationErrors`.

The interpreter implementation offers mechanisms to allow a caller to be unaware if a particular function invocation leads to bytecode interpretation or is executed directly at interpreter-level. The two basic kinds of Gateway classes expose either an interpreter-level function to application-level execution (`interp2app`) or allow transparent invocation of application-level helpers (`app2interp`) at interpreter-level.

Another task of the bytecode interpreter is to care for exposing its basic code, frame, module and function objects to application-level code. Such runtime introspection and modification abilities are implemented via interpreter descriptors (also see Raymond Hettingers how-to guide for descriptors in Python, PyPy uses this model extensively).

A significant complexity lies in function argument parsing. Python as a language offers flexible ways of providing and receiving arguments for a particular function invocation. It does not only take special care to get this right, but it also presents difficulties for the annotation pass which performs a whole-program analysis on the bytecode interpreter, argument parsing and gatewaying code in order to infer the types of all values flowing across function calls.

For this reason PyPy resorts to generate specialized frame classes and functions at initialization time in order to let the annotator only see rather static program flows with homogenous name-value assignments on function invocations.

7.2 Bytecode Interpreter Implementation Classes

7.2.1 Frame classes

The concept of Frames is pervasive in executing programs and on virtual machines in particular. They are sometimes called *execution frame* because they hold crucial information regarding the execution of a Code object, which in turn is often directly related to a Python Function. Frame instances hold the following state:

- the local scope holding name-value bindings, usually implemented via a "fast scope" which is an array of wrapped objects
- a blockstack containing (nested) information regarding the control flow of a function (such as `while` and `try` constructs)
- a value stack where bytecode interpretation pulls object from and puts results on.

- a reference to the *globals* dictionary, containing module-level name-value bindings
- debugging information from which a current line-number and file location can be constructed for tracebacks

Moreover the Frame class itself has a number of methods which implement the actual bytecodes found in a code object. In fact, PyPy already constructs four specialized Frame class variants depending on the code object:

- PyInterpFrame (in *pypy/interpreter/pyopcode.py*) for basic simple code objects (not involving generators or nested scopes)
- PyNestedScopeFrame (in *pypy/interpreter/nestedscope.py*) for code objects that reference nested scopes, inherits from PyInterpFrame
- PyGeneratorFrame (in *pypy/interpreter/generator.py*) for code objects that yield values to the caller, inherits from PyInterpFrame
- PyNestedScopeGeneratorFrame for code objects that reference nested scopes and yield values to the caller, inherits from both PyNestedScopeFrame and PyGeneratorFrame

7.2.2 Code Class

PyPy's code objects contain the same information found in CPython's code objects. They differ from Function objects in that they are only immutable representations of source code and don't contain execution state or references to the execution environment found in *Frames*. Frames and Functions have references to a code object. Here is a list of Code attributes:

- `co_flags` flags if this code object has nested scopes/generators
- `co_stacksize` the maximum depth the stack can reach while executing the code
- `co_code` the actual bytecode string
- `co_argcount` number of arguments this code object expects
- `co_varnames` a tuple of all argument names pass to this code object
- `co_nlocals` number of local variables
- `co_names` a tuple of all names used in the code object
- `co_consts` a tuple of prebuilt constant objects ("literals") used in the code object
- `co_cellvars` a tuple of Cells containing values for access from nested scopes
- `co_freevars` a tuple of Cell names from "above" scopes
- `co_filename` source file this code object was compiled from
- `co_firstlineno` the first linenumber of the code object in its source file
- `co_name` name of the code object (often the function name)
- `co_lnotab` a helper table to compute the line-numbers corresponding to bytecodes

In PyPy, code objects also have the responsibility for creating their Frame objects via the *'create_frame()'* method. With proper support of parser and compiler this should allow to create custom Frame objects extending the execution of functions in various ways. The several Frame classes already utilize this flexibility in order to implement Generators and Nested Scopes.

7.2.3 Function and Method Classes

The PyPy `Function` class (in *pypy/interpreter/function.py*) represents a Python function. A `Function` carries the following main attributes:

- `func_doc` the docstring (or None)
- `func_name` the name of the function
- `func_code` the Code object representing the function source code
- `func_defaults` default values for the function (built at function definition time)
- `func_dict` dictionary for additional (user-defined) function attributes
- `func_globals` reference to the globals dictionary
- `func_closure` a tuple of Cell references

`Functions` classes also provide a `__get__` descriptor which creates a Method object holding a binding to an instance or a class. Finally `Functions` and `Methods` both offer a `call_args()` method which executes the function given an Arguments class instance.

7.2.4 Arguments Class

The Argument class (in *pypy/interpreter/argument.py*) is responsible for parsing arguments passed to functions. Python has rather complex argument-passing concepts:

- positional arguments
- keyword arguments specified by name
- default values for positional arguments, defined at function definition time
- "star args" which allow a function to accept remaining positional arguments
- "star keyword args" allow a function to accept additional arbitrary name-value bindings

Moreover, a Function object can get bound to a class or instance. In this case the first argument to the underlying function becomes the bound object. The `Arguments` provides means to allow all this argument parsing and also cares for error reporting.

7.2.5 Module Class

A `Module` instance represents execution state usually constructed from executing the module's source file. In addition to such a module's global `__dict__` dictionary it has the following application level attributes:

- `__doc__` the docstring of the module
- `__file__` the source filename from which this module was instantiated
- `__path__` state used for relative imports

Apart from the basic Module used for importing application-level files there is a more refined `MixedModule` class (see *pypy/interpreter/mixedmodule.py*) which allows to define name-value bindings both at application level and at interpreter level. See the `__builtin__` module's *pypy/module/__builtin__/__init__.py* file for an example and the higher level chapter on Modules in the coding guide.

7.2.6 Gateway Classes

A unique PyPy property is the ability to cross easily the barrier between interpreted and machine-level code (often refered to as the difference between interpreter-level and application-level). Be aware that the according code (in *pypy/interpreter/gateway.py*) for crossing the barrier in both directions is somewhat involved, mostly due to the fact that the type-infering annotator needs to keep track of the types of objects flowing across those barriers.

7.2.7 Making Interpreter-level Functions Available at Application-level

In order to make an interpreter-level function available at application level one invokes `pypy.interpreter.g` Such a function usually takes a `space` argument and any number of positional arguments. Additionally, such functions can define an `unwrap_spec` telling the `interp2app` logic how application-level provided arguments should be unwrapped before the actual interpreter-level function is invoked. For example interpreter descriptors such as the `Module.__new__` method for allocating and constructing a Module instance are defined with such code:

```
    Module.typedef = TypeDef("module",
        __new__ = interp2app(Module.descr_module__new__.im_func,
                             unwrap_spec=[ObjSpace, W_Root, Arguments]),
        __init__ = interp2app(Module.descr_module__init__),
                        # module dictionaries are readonly attributes
        __dict__ = GetSetProperty(descr_get_dict, cls=Module),
        __doc__ = 'module(name[, doc])\n\nCreate a module object...'
        )
```

The actual `Module.descr_module__new__` interpreter-level method referenced from the `__new__` keyword argument above is defined like this:

```
    def descr_module__new__(space, w_subtype, __args__):
        module = space.allocate_instance(Module, w_subtype)
        Module.__init__(module, space, None)
        return space.wrap(module)
```

Summarizing, the `interp2app` mechanism takes care to route an application level access or call to an internal interpreter-level object appropriately to the descriptor, providing enough precision and hints to keep the type-infering annotator happy.

7.2.8 Calling into Application Level Code from Interpreter-Level

Application level code is often preferable. Therefore, we often like to invoke application level code from interpreter-level. This is done via the Gateway's `app2interp` mechanism which we usually invoke at definition time in a module. It generates a hook which looks like an interpreter-level function accepting a space and an arbitrary number of arguments. When calling a function at interpreter-level the caller side does usually not need to be aware if its invoked function is run through the PyPy interpreter or if it will directly execute on the machine (after translation).

Here is an example showing how we implement the Metaclass finding algorithm of the Python language in PyPy:

```
    app = gateway.applevel(r'''
        def find_metaclass(bases, namespace, globals, builtin):
            if '__metaclass__' in namespace:
                return namespace['__metaclass__']
            elif len(bases) > 0:
                base = bases[0]
                if hasattr(base, '__class__'):
                        return base.__class__
                else:
                        return type(base)
            elif '__metaclass__' in globals:
                return globals['__metaclass__']
            else:
                try:
                    return builtin.__metaclass__
                except AttributeError:
                    return type
    ''', filename=__file__)

    find_metaclass  = app.interphook('find_metaclass')
```

The `find_metaclass` interpreter-level hook is invoked with five arguments from the `BUILD_CLASS` opcode implementation in *pypy/interpreter/pyopcode.py*:

```
    def BUILD_CLASS(f):
        w_methodsdict = f.valuestack.pop()
        w_bases       = f.valuestack.pop()
        w_name        = f.valuestack.pop()
        w_metaclass = find_metaclass(f.space, w_bases,
                                     w_methodsdict, f.w_globals,
                                     f.space.wrap(f.builtin))
        w_newclass = f.space.call_function(w_metaclass, w_name,
                                           w_bases, w_methodsdict)
        f.valuestack.push(w_newclass)
```

Note that at a later point we can rewrite the `find_metaclass` implementation at interpreter-level and we would not have to modify the calling side at all.

7.2.9 Introspection and Descriptors

Python traditionally has a very far-reaching introspection model for bytecode interpreter related objects. In PyPy and in CPython read and write accesses to such objects are routed to descriptors. Of course in CPython those are implemented in `C` while in PyPy they are implemented in interpreter-level Python code.

All instances of a Function, Code, Frame or Module classes are also `Wrappable` instances which means that they can be represented at application level. These days, a PyPy object space needs to work with a basic descriptor lookup when it encounters accesses to an interpreter-level object: an object space asks a wrapped object for its type via a `getclass` method and then calls the type's `lookup(name)` function in order to receive a descriptor function. Most of PyPy's internal object descriptors are defined at the end of *pypy/interpreter/typedef.py*. You can use these definitions as a reference for the exact attributes of interpreter classes visible at application level.

8 PyPy - Object Spaces

8.1 Introduction

This document describes aspects of the implementation of PyPy's Object Spaces.

8.2 Object Space Interface

This is a draft version of the Object Space interface. It is still evolving, although the public interface is not evolving as much as the internal interface that subclasses need to know about. This document now generally refers to the public interface; for subclassing you need to poke at the code in detail anyway.

8.2.1 Administrative Functions

initialize(): Function which initializes w_builtins and the other w_constants.

getexecutioncontext(): Return current active execution context.

8.2.2 Operations on Objects in ObjSpace

These functions both take and return "wrapped" objects.

The following functions implement the same operations as those in CPython:

`id, type, issubtype, iter, repr, str, len, hash,`

`getattr, setattr, delattr, getitem, setitem, delitem,`

`pos, neg, not_, abs, invert, add, sub, mul, truediv, floordiv, div, mod, divmod, pow, lshift, rshift, and_, or_, xor,`

`hex, oct, int, float, ord,`

`lt, le, eq, ne, gt, ge, contains,`

`inplace_add, inplace_sub, inplace_mul, inplace_truediv, inplace_floordiv, inplace_div, inplace_mod, inplace_pow, inplace_lshift, inplace_rshift, inplace_and, inplace_or, inplace_xor,`

`get, set, delete`

next(w): Calls the next function for iterator w, or raises a real NoValue.

call(w_callable, w_args, w_kwds): Calls a function with the given args and keywords.

call_function(w_callable, *args_w, **kw_w): Convenience function that collects the arguments in a wrapped tuple and dict and invokes 'call()'.

is_(w_x, w_y): Implements 'w_x is w_y'. (Returns a wrapped result too!)

isinstance(w_obj, w_type): Implements 'issubtype(type(w_obj), w_type)'. (Returns a wrapped result too!)

exception_match(w_exc_type, w_check_class): Checks if the given exception type matches 'w_check_class'. Used in matching the actual exception raised with the list of those to catch in an except clause. (Returns a wrapped result too!)

8.2.3 Creation of Application Level Objects

wrap(x): Returns a wrapped object that is a reference to the interpreter-level object x. This can be used either on simple immutable objects (integers, strings...) to create a new wrapped object, or on complex mutable objects to obtain an application-level-visible reference to them (e.g. instances of internal bytecode interpreter classes).

newbool(b): Creates a Bool Object from an interpreter level object.

newtuple([..]): Takes an interpreter level list of wrapped objects.

newlist([..]): Takes an interpreter level list of wrapped objects.

newdict([..]): Takes an interpreter level list of interpreter level pairs of wrapped key: wrapped value entries (and NOT an interpreter level dictionary!).

newslice(w_start, w_end, w_step): Makes a new slice object.

newstring(asciilist): Creates a string from a list of wrapped integers.

newunicode(codelist): Creates a unicode string from a list of integers.

8.2.4 Conversions from Application Level to Interpreter Level

unwrap(w_x): Return Interpreter Level equivalent of w_x

interpclass_w(w_x): If w_x is a wrapped instance of a bytecode interpreter class - for example Function, Frame, Cell, etc. - the return it unwrapped. Otherwise return None.

int_w(w_x): If w_x is an application-level integer or long which can be converted without overflow to an integer, return an interpreter-level integer. Otherwise raise TypeError or OverflowError.

str_w(w_x): If w_x is an application-level string, return an interpreter-level string. Otherwise raise TypeError.

float_w(w_x): If w_x is an application-level float, integer or long, return interpreter-level float. Otherwise raise TypeError or OverflowError in case of very large longs.

is_true(w_x): Return a interpreter level bool (True or False).

unpackiterable(w_iterable, expected_length=None): Unpack an iterable object into a real (interpreter level) list. Raise a real ValueError if the expected_length is wrong.

unpacktuple(w_tuple, expected_length=None): Same as unpackiterable(), but only for tuples.

8.2.5 Data Members

- self.builtin
- self.sys
- self.w_None: The ObjSpace's None
- self.w_True: The ObjSpace's True

- self.w_False: The ObjSpace's False
- self.w_Ellipsis: The ObjSpace's Ellipsis
- self.w_NotImplemented: The ObjSpace's NotImplemented
- **ObjSpace.MethodTable:** List of tuples (method name, symbol, number of arguments, list of special names) for the regular part of the interface. (Tuples are interpreter level.)
- **ObjSpace.BuiltinModuleTable:** List of names of built-in modules.
- **ObjSpace.ConstantTable:** List of names of the constants that the object space should define
- **ObjSpace.ExceptionTable:** List of names of exception classes.

8.3 The Standard Object Space

8.3.1 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 RestrictedPython. 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 did we make such an effort 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 that.

In our case it is a later optimization that we could make. We just don't want to make it now (and certainly not hard-coded at this level -- it could be introduced by the code generators at translation time). 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.

Note that the Standard Object Space implementation uses MultiMethod dispatch instead of the complex rules of "Object/abstract.c". This can probably be translated to a different low-level dispatch implementation that would be binary compatible with CPython's (basically the PyTypeObject structure and its function pointers). If compatibility is not required it will be more straightforwardly converted into some efficient multimethod code.

8.3.2 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 is called `xxxtype.py` for a type `xxx`;
- 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 separate the Python type object that are visible to the user and the actual implementation of its instances cleany. It is possible (though not done so far) to provide *several* implementations of the instances of the same Python type. The `__new__()` method could decide to create one or the other. From the user's point of view, they are still all instances of exactly the same type; the possibly multiple internal `W_XxxObject` classes are not visible. PyPy knows that (e.g.) the application-level type of its interpreter-level `W_TupleObject` instances is "tuple" because there is a `typedef` class attribute in `W_TupleObject` which points back to the tuple type specification from tupletype.py.

8.4 The Trace Object Space

The Trace Object Space was first written at the Amsterdam sprint. The ease of implementation of the Trace Object Space in *pypy/objspace/trace.py* underlines the power of the Object Space abstraction. Effectively it is a simple proxy object space. It has gone through various refactors to reach its original objective, which was to show how bytecode in code objects ultimately performs computation via an object space.

This space will intercept space operations in realtime and as a side effect will memorize them. It also traces frame creation, deletion and bytecode execution. Its implementation delegates to another object space - usually the standard object space - in order to carry out the operations.

The pretty printing aims to be a graphical way of introducing programmers, and especially ones familiar with CPython, to how PyPy works from a bytecode and frames perspective. As a result one can grasp an intuitive idea of how Abstract Interpretation records via tracing all execution paths of the individual operations if one removes the bytecode out of the equation. This is the purpose of the Flow Object Space.

Another educational use of Trace Object Space is that it allows a Python user who has little understanding of how the interpreter works, a rapid way of understanding what bytecodes are and what an object space is. When a statement or expression is typed on the command line, one can see what is happening behind the scenes. This will hopefully give users a better mental framework when they are writing code.

To make use of the tracing facilities you can at runtime switch your interactive session to tracing mode by typing:

```
>>> __pytrace__ = 1
```

Note that tracing mode will not show or record all space operations by default to avoid presenting too much information. Usually only non-helper operations are shown.

A quick introduction on how to use the trace object space can be found here. A number of options for configuration is here in traceconfig.py.

8.5 The Thunk Object Space

This small object space, meant as a nice example, wraps another object space (e.g. the standard one) and adds two capabilities: lazily computed objects (computed only when an operation is performed on them), and "become", which completely and globally replaces an object with another.

Example usage:

```
$ py.py -o thunk
>>>> def f():
...     print 'computing...'
...     return 6*7
...
>>>> x = thunk(f)
>>>> x
computing...
42
>>>> x
42
>>>> y = thunk(f)
>>>> type(y)
computing...
<pypy type 'int'>
```

8.6 The Flow Object Space

8.6.1 Introduction

The task of the FlowObjSpace is to generate a control-flow graph from a function. This graph will also contain a trace of the individual operations, so that it is actually just an alternate representation for the function.

The FlowObjSpace is an object space, which means that it exports the standard object space interface and it is driven by the bytecode interpreter.

The basic idea is that if the bytecode interpreter is given a function, e.g.:

```
def f(n):
  return 3*n+2
```

it will do whatever bytecode dispatching and stack-shuffling needed, during which it issues a sequence of calls to the object space. The FlowObjSpace merely records these calls (corresponding to "operations") in a structure called a basic block. To track which value goes where, the FlowObjSpace invents placeholder "wrapped objects" and gives them to the interpreter, so that they appear in some next operation. This technique is an example of Abstract Interpretation.

For example, if the placeholder `v1` is given as the argument to the above function, the bytecode interpreter will call the following operations:

```
v2 = space.mul(space.wrap(3), v1)
v3 = space.add(v2, space.wrap(2))
```

and return `v3` as the result. During these calls the FlowObjSpacer will record a basic block:

```
Block(v1):     # input argument
  v2 = mul(Constant(3), v1)
  v3 = add(v2, Constant(2))
```

8.6.2 The Flow Model

`pypy.objspace.flow.model` defines the data model used by the flow graphs, as created by the Flow Object Space, manipulated by the modules such as `simplify` and `transform` in `pypy.translator`, and in general read by almost all the modules in `pypy.translator`.

It is recommended to play with `python translator.py` on a few examples to get an idea of the structure of flow graphs. Here is a short summary of the non-obvious parts.

FunctionGraph A container for one graph (corresponding to one function).

- **startblock:** the first block. This is where the control goes when the function is called. The input arguments of the startblock are the function's arguments. If the function takes a `*args` argument, the `args` tuple is given as the last input argument of the startblock.
- **returnblock:** the (unique) block that performs a function return. It is empty, not actually containing any `return` operation; the return is implicit. The returned value is the unique input variable of the returnblock.
- **exceptblock:** the (unique) block that raises an exception out of the function. The two input variables are the exception class and the exception value, respectively. (No other block will actually link to the exceptblock if the function does not explicitely raise exceptions.)

Block A basic block, containing a list of operations and ending in jumps to other basic blocks. All the values that are "live" during the execution of the block are stored in Variables. Each basic block uses its own distinct Variables.

inputargs: list of fresh, distinct Variables that represent all the values that can enter this block from any of the previous blocks.

operations: list of SpaceOperations.

exitswitch: see below

exits: list of Links representing possible jumps from the end of this basic block to the beginning of other basic blocks.

Each Block ends in one of the following ways:

- unconditional jump: exitswitch is None, exits contains a single Link.
- conditional jump: exitswitch is one of the Variables that appear in the Block, and exits contains one or more Links (usually 2). Each Link's exitcase gives a concrete value. This is the equivalent of a "switch": the control follows the Link whose exitcase matches the run-time value of the exitswitch Variable. It is a run-time error if the Variable doesn't match any exitcase. (Currently only used with 2 Links whose exitcase are False and True, respectively.)
- exception catching: exitswitch is `Constant(last_exception)`. The first Link has exitcase set to None and represents the non-exceptional path. The following Links have exitcase set to a subclass of Exception, and are taken when the *last* operation of the basic block raises a matching exception. (Thus the basic block must not be empty, and only the last operation is protected by the handler.)
- return or except: the returnblock and the exceptblock have operations set to an empty tuple, exitswitch to None, and exits empty.

Link A link from one basic block to another.

prevblock: the Block that this Link is an exit of.

target: the target Block this Link points to.

args: a list of Variables and Constants, of the same size as the target Block's inputargs, which gives all the values passed into the next block. (Note that each Variable used in the prevblock may appear zero, one or more times in the `args` list.)

exitcase: see above.

last_exception: None or a Variable; see below.

last_exc_value: None or a Variable; see below.

Note that `args` uses Variables from the prevblock, which are matched to the target block's `inputargs` by position as in a tuple assignment or function call would do.

If the link is an exception-catching one, the `last_exception` and `last_exc_value` are set to two fresh Variables that are considered to be created when the link is entered; at run-time they will hold the exception class and value, respectively. These two new variables can only be used in the same link's `args` list, to be passed to the next block (as usual, they may actually not appear at all, or appear several times in `args`).

SpaceOperation A recorded (or otherwise generated) basic operation.

opname: the name of the operation. Generally one from the list in `pypy.interpreter.baseobjspa`

args: list of arguments. Each one is a Constant or a Variable seen previously in the basic block.

result: a *new* Variable into which the result is to be stored.

Note that operations usually cannot implicitly raise exceptions at run-time; so for example code generators can assume that a `getitem` operation on a list is safe and can be performed without bound checking. The exceptions to this rule are: (1) if the operation is the last in the block, which ends with `exitswitch == Constant(last_exception)` the implicit exceptions must be checked for, generated, and caught appropriately; (2) calls to other functions, as per `simple_call` or `call_args`, can always raise whatever the called function can raise --- and this kind of exceptions must be passed through to the parent unless they are caught as above.

Variable A placeholder for a run-time value. There is mostly debugging stuff here.

name: it is good style to use the Variable object itself instead of its `name` attribute to reference a value, although the `name` is guaranteed to be unique.

Constant A constant value used as argument to a SpaceOperation, or as value to pass across a Link to initialize an input Variable in the target Block.

value: the concrete value represented by this Constant.
key: a hashable object representing the value.

A Constant can occasionally store a mutable Python object. It represents a static, pre-initialized, read-only version of that object. The flow graph should not attempt actually to mutate such Constants.

8.6.3 How the FlowObjSpace Works

The FlowObjSpace works by recording all operations issued by the bytecode interpreter into basic blocks. A basic block ends in one of two cases: if the bytecode interpreters calls `is_true()`, or if a joinpoint is reached.

- A joinpoint occurs if the next operation is about to be recorded into the current block, but there is already another block that records an operation for the same bytecode position. This means that the bytecode interpreter has closed a loop and is interpreting already-seen code again. In this situation, we interrupt the bytecode interpreter and we make a link from the end of the current block back to the previous block, thus closing the loop in the flow graph as well. (Note that this occurs only if an operation is about to be recorded which allows some amount of constant-folding.)
- If the bytecode interpreter calls `is_true()`, the FlowObjSpace doesn't generally know if the answer should be True or False, so it puts a conditional jump and generates two successor blocks for the current basic block. There is some trickery involved so that the bytecode interpreter is fooled into thinking that `is_true()` first returns False (and the subsequent operations are recorded in the first successor block), and later the *same* call to `is_true()` also returns True (and the subsequent operations go to the other successor block this time).

9 PyPy - Getting Started

9.1 Just the Facts

9.1.1 Getting & Running the PyPy 0.7 Release

Download one of the following release files and unpack it:

pypy-0.7

- download one of
  - pypy-0.7.0.tar.bz2 (unix line endings) or
  - pypy-0.7.0.tar.gz (unix line endings) or
  - pypy-0.7.0.zip (windows line-endings) and unpack it

Unpack the package and change directory to `pypy-0.7.0` and execute the following command line:

```
python pypy/bin/py.py
```

This will give you a PyPy prompt, i.e. a very compliant Python interpreter implemented in Python. PyPy passes around 90% of CPythons core language regression tests. Because this invocation of PyPy still runs on top of CPython, it runs around 2000 times slower than the original CPython.

However, since the 0.7.0 release it is possible to use PyPy to translate itself to lower level languages after which it runs standalone, is not dependant on CPython anymore and becomes faster.

9.1.2 Running all of PyPy's tests

If you want to see if PyPy works on your machine/platform you can simply run PyPy's large test suite with:

```
cd pypy
python test_all.py
```

test_all.py is just another name for py.test which is the testing tool that we are using and enhancing for PyPy.

9.1.3 Filing Bugs or Feature Requests

You may file bug reports on our issue tracker which is also accessible through the 'issues' top menu of the PyPy website. using the development tracker has more detailed information on specific features of the tracker.

9.2 Interesting Starting Points in PyPy

The following assumes that you have successfully downloaded and extracted the PyPy release or have checked out PyPy using svn. It assumes that you are in the top level directory of the PyPy source tree, e.g. pypy-x.x (if you got a release) or pypy-dist (if you checked out the most recent version using subversion).

9.2.1 Main Entry Point

9.2.2 The py.py Interpreter

To start interpreting Python with PyPy, use Python 2.3 or greater:

```
cd pypy/bin
python py.py
```

After a few seconds (remember: this is running on top of CPython), you should be at the PyPy prompt, which is the same as the Python prompt, but with an extra ">".

Now you are ready to start running Python code. Most Python modules should work if they don't involve CPython extension modules. Here is an example of determining PyPy's performance in pystones:

```
>>>> from test import pystone
>>>> pystone.main(10)
```

The parameter is the number of loops to run through the test. The default is 50000, which is far too many to run in a reasonable time on the current PyPy implementation.

9.2.3 py.py Options

To list the PyPy interpreter command line options, type:

```
cd pypy/bin
python py.py --help
```

As an example of using PyPy from the command line, you could type:

```
python py.py -c "from test import pystone; pystone.main(10)"
```

Alternatively, as with regular Python, you can simply give a script name on the command line:

```
python py.py ../../lib-python/2.4.1/test/pystone.py 10
```

9.2.4 Special PyPy Features

9.2.5 Interpreter-level Console

There are a few extra features of the PyPy console: If you press <Ctrl-C> on the console you enter the interpreter-level console, a usual CPython console. You can then access internal objects of PyPy (e.g. the object space) and any variables you have created on the PyPy prompt with the prefix w_:

```
>>>> a = 123
>>>> <Ctrl-C>
*** Entering interpreter-level console ***
>>> w_a
W_IntObject(123)
```

Note that the prompt of the interpreter-level console is only '>>>' since it runs on CPython level. If you want to return to PyPy, press <Ctrl-D> (under Linux) or <Ctrl-Z>, <Enter> (under Windows).

You may be interested in reading more about the distinction between interpreter-level and app-level.

9.2.6 Tracing Bytecode and Operations on Objects

You can use the trace object space to monitor the interpretation of bytecodes in connection with object space operations. To enable it, set `__pytrace__`=1 on the interactive PyPy console:

```
>>>> __pytrace__ = 1
Tracing enabled
>>>> a = 1 + 2
|- <<<< enter <inline>a = 1 + 2 @ 1 >>>>
|- 0    LOAD_CONST    0 (W_IntObject(1))
|- 3    LOAD_CONST    1 (W_IntObject(2))
|- 6    BINARY_ADD
  |-    add(W_IntObject(1), W_IntObject(2))   -> W_IntObject(3)
|- 7    STORE_NAME    0 (a)
  |-    setitem(W_DictObject([<Entry 3098577608L,W_StringObject('__b...
        -> <W_NoneObject()>
|-10    LOAD_CONST    2 (<W_NoneObject()>)
|-13    RETURN_VALUE
|- <<<< leave <inline>a = 1 + 2 @ 1 >>>>
```

9.2.7 Lazily Computed Objects

One of the original features provided by PyPy is the "thunk" object space, providing lazily-computed objects in a fully transparent manner:

```
cd pypy/bin
python py.py -o thunk

>>>> def longcomputation(lst):
....     print "computing..."
....     return sum(lst)
....
>>>> x = thunk(longcomputation, range(5))
>>>> y = thunk(longcomputation, range(10))
```

from the application perspective, x and y represent exactly the objects being returned by the `longcomputation()` invocations. You can put these objects into a dictionary without triggering the computation:

```
>>>> d = {5: x, 10: y}
>>>> result = d[5]
>>>> result
computing...
```

```
10
>>>> type(d[10])
computing...
<type 'int'>
>>>> d[10]
45
```

It is interesting to note that this lazy-computing Python extension is solely implemented in a small *objspace/thunk.py* file consisting of around 100 lines of code. Since the 0.8.0 release it is even possible to translate PyPy with the thunk object space.

9.2.8 Running the Tests

The PyPy project uses test-driven-development. Right now, there are a couple of different categories of tests which you can run. To run all the unit tests:

```
cd pypy
python test_all.py
```

Alternatively, you may run subtests by going to the correct subdirectory and running them individually:

```
python test_all.py interpreter/test/test_pyframe.py
```

`test_all.py` actually is just a synonym for py.test which is our external testing tool. If you have installed that you can as well just issue `py.test DIRECTORY_OR_FILE` in order to perform test runs or simply start it without arguments to run all tests below the current directory.

Finally, there are the CPython regression tests which you can run like this (this will take hours and hours and hours):

```
cd lib-python/2.4.1/test
python ../../../pypy/test_all.py
```

or if you have installed py.test then you simply say:

```
py.test
```

from the lib-python/2.4.1/test directory. You need to have a checkout of the testresult directory. Running one of the above commands tells you how to proceed.

9.2.9 Demos

The *demo/* directory contains examples of various aspects of PyPy, ranging from running regular Python programs (that we used as compliance goals) over experimental distribution mechanisms to examples translating sufficiently static programs into low level code.

9.2.10 Trying out the Translator

The translator is a tool based on the PyPy interpreter which can translate sufficiently static Python programs into low-level code. To be able to use it you need to:

- Download and install Pygame if you do not already have it.
- Have an internet connection. The flowgraph viewer connects to codespeak.net and lets it convert the flowgraph with a patched version of Dot Graphviz that does not crash. This is only needed if you want to look at the flowgraphs.

To start the interactive translator do:

```
cd pypy/bin
python translator.py
```

Test snippets of translatable code are provided in the file `pypy/translator/test/snippet.py`, which is imported under the name `test`. For example:

```
>>> t = Translator(test.is_perfect_number)
>>> t.view()
```

After that, the graph viewer pops up, that lets you interactively inspect the flowgraph. To move around, click on something that you want to inspect. To get help about how to use it, press 'H'. To close it again, press 'Q'.

9.2.11 Trying out the Type Annotator

We have a type annotator that can completely infer types for functions like `is_perfect_number` (as well as for much larger examples):

```
>>> a = t.annotate([int])
>>> t.view()
```

Move the mouse over variable names (in red) to see their inferred types. To perform simplifications based on the annotation you can do:

```
>>> a.simplify()
```

9.2.12 Translating the Flow Graph to C Code

The graph can be turned into C code:

```
>>> t.specialize()
>>> f = t.ccompile()
```

The first command replaces the operations by other low level versions that only use low level types that are available in C (e.g. int). This can also be ommited although it is not recommended. To try out the compiled version:

```
>>> f(5)
False
>>> f(6)
True
```

9.2.13 Translating the Flow Graph to LLVM Code

To translate for LLVM (low level virtual machine) you must first have LLVM installed with the CVS version - the how to install LLVM provides some helpful hints. Please note that you do not need the CFrontend to compile, make tools-only.

The LLVM backend is still in an experimental state. However, it is very close to the functionality of the C backend. Calling compiled LLVM code from CPython is more restrictive than the C backend - the return type and the arguments of the entry function must be ints, floats or bools. The emphasis of the LLVM backend is to compile standalone executables - please see the pypy/translator/llvm/demo directory for examples.

Here is a simple example to try:

```
>>> t = Translator(test.my_gcd)
>>> a = t.annotate([int, int])
>>> t.specialize()
>>> print t.llvm()

<...  really huge amount of LLVM code  ...>

>>> f = t.llvmcompile()
>>> f.pypy_my_gcd_wrapper(15, 10)
5
```

9.2.14 A Slightly Larger Example

There is a small-to-medium demo showing the translator and the annotator:

```
cd demo
python bpnn.py
```

This causes `bpnn.py` to display itself as a call graph and class hierarchy. Clicking on functions shows the flow graph of the particular function. Clicking on a class shows the attributes of its instances. All this information (call graph, local variables' types, attributes of instances) is computed by the annotator.

As soon as you close the PyGame window, the function is turned into C code, compiled and executed.

9.2.15 Translating the PyPy Interpreter

Not for the faint of heart nor the owner of a very old machine: you can translate the whole of PyPy to low level C code. This is the largest and ultimate example of source that our translation toolchain can process:

```
cd pypy/translator/goal
python translate_pypy.py --run
```

By default the translation process will try to use the Boehm-Demers-Weiser garbage collector for the translated PyPy (Use `--gc=ref` to use our own reference counting implementation which at the moment is slower but doesn't have external dependencies).

This whole process will take some time and quite a lot of memory. To reduce the memory footprint of the translation process you can use the option `--lowmem`. With this option the whole process should be runnable on a machine with 512Mb of RAM. If the translation has finished running and after you closed the graph you will be greeted (because of `--run` option) by the friendly prompt of a PyPy executable that is not running on top of CPython any more:

```
[translation:info] created: ./pypy-c
[translation:info] Running compiled c source...
debug: entry point starting
debug:  argv -> ./pypy-c
debug: importing code
debug: calling code.interact()
Python 2.4.1 (pypy 0.7.1 build 18929) on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>> 1 + 1
2
>>>>
```

With the default options, you can find the produced executable under the name `pypy-c`. Type `pypy-c --help` to see the options it supports - mainly the same basic options of CPython. In addition, `pypy-c --info` prints the translation options that where used to produce this particular executable. This executable contains a lot of things that are hard-coded for your particular system (including paths), so it's not really meant to be installed or redistributed at the moment.

If you exit the interpreter you get a pygame window with all the flowgraphs plus a pdb prompt. Moving around in the resulting flow graph is difficult because of the sheer size of the result. For this reason the debugger prompt you get at the end has been enhanced with commands to facilitate locating functions and classes. Type `help graphs` for a list of the new commands. Help is also available on each of these new commands.

The `translate_pypy` script itself takes a number of options controlling what and how to translate. See `translate_pypy.py -h`. Some of the more interesting options are:

- `--text`: don't show the flowgraph after the translation is done. This is useful if you don't have pygame installed.
- `--gc=boehm|ref`: choose between using the Boehm-Demers-Weiser garbage collector or our own reference counting implementation (as we have seen Boehm's collector is the default).

You can also use the translate_pypy.py script to try out several smaller programs, e.g. a slightly changed version of Pystone:

```
cd pypy/translator/goal
python translate_pypy.py targetrpystone
```

9.2.16 Translating with the Thunk Object Space

It is also possible to experimentally translate a PyPy version using the "thunk" object space:

```
cd pypy/translator/goal
python translate_pypy.py --run targetthunkstandalone
```

the examples in lazily computed objects should work in the translated result.

9.2.17 Translating using the LLVM Backend

To create a standalone executable using the experimental LLVM compiler infrastructure:

```
./run_pypy-llvm.sh
```

9.3 Where to Start Reading the Sources

PyPy is made from parts that are more or less independent from each other. You should start looking at the part that attracts you most (all parts are relative to the PyPy toplevel directory). You may look at our directory reference or start off at one of the following points:

- *pypy/interpreter* contains the bytecode interpreter: bytecode dispatcher in pyopcode.py, frame and code objects in eval.py and pyframe.py, function objects and argument passing in function.py and argument.py, the object space interface definition in baseobjspace.py, modules in module.py and mixedmodule.py. Core types supporting the bytecode interpreter are defined in typedef.py.
- *pypy/interpreter/pyparser* contains a recursive descent parser, and input data files that allow to parse both Python 2.3 and 2.4 syntax. Once the input data has been processed, the parser can be translated by the above machinery into efficient code.
- *pypy/interpreter/astcompiler* contains the compiler. This contains a modified version of the compiler package from CPython that fixes some bugs and is translatable. It is new in 0.8.0 that the compiler and parser are translatable and it makes using the resulting binary interactively much more pleasant.
- *pypy/objspace/std* contains the Standard object space. The main file is objspace.py. For each type, the files `xxxtype.py` and `xxxobject.py` contain respectively the definition of the type and its (default) implementation.
- *pypy/objspace* contains a few other object spaces: the thunk, trace and flow object spaces. The latter is a relatively short piece of code that builds the control flow graphs when the bytecode interpreter runs in it.
- *pypy/translator* contains the code analysis and generation stuff. Start reading from translator.py, from which it should be easy to follow the pieces of code involved in the various translation phases.
- *pypy/annotation* contains the data model for the type annotation that can be inferred about a graph. The graph "walker" uses this is in *pypy/translator/annrpython.py*.
- *pypy/rpython* contains the code of the RPython typer. The typer transforms annotated flow graphs in a way that makes them very similar to C code so that they can be easily translated. The graph transformations are controlled by the stuff in *pypy/rpython/rtyper.py*. The object model that is used can be found in *pypy/rpython/lltypesystem/lltype.py*. For each RPython type there is a file rxxxx.py that contains the low level functions needed for this type.

9.4 Additional Tools for running (and hacking) PyPy

We use some optional tools for developing PyPy. They are not required to run the basic tests or to get an interactive PyPy prompt but they help to understand and debug PyPy especially for the ongoing translation work.

9.4.1 Graphviz & PyGame for Flowgraph Viewing (Highly Recommended)

Graphviz and PyGame are both neccessary if you want to look at generated flowgraphs:

Graphviz: http://www.research.att.com/sw/tools/graphviz/download.html
PyGame: http://www.pygame.org/download.shtml

9.4.2 CLISP

The CLISP backend is optional and not quite uptodate with the rest of PyPy. There are still a few examples you can try our backend out on. Here is a link to a LISP implementation that should basically work:

http://clisp.cons.org/

9.4.3 py.test and the py-lib

The py library is used to support the PyPy development and running our tests against code and documentation as well as compliancy tests. You don't need to install the py library because it ships with PyPy and *pypy/test_all.py* is an alias for `py.test`. But if you want to have the `py.test` tool generally in your path, you might like to visit:

http://codespeak.net/py/current/doc/getting-started.html

10 PyPy - Coding Guide

This document describes coding requirements and conventions for working with the PyPy code base. Please read it carefully and ask back any questions you might have.

10.1 Restricted Python

We are writing a Python interpreter in Python, using Python's well known ability to step behind the algorithmic problems as language. At first glance one might think this achieves nothing but a better understanding how the interpreter works. This alone would make it worth doing, but we have much larger goals.

10.2 CPython vs. PyPy

Compared to the CPython implementation, Python takes the role of the C Code. We rewrite the CPython interpreter in Python itself. We could also aim at writing a more flexible interpreter at C level but we want to use Python to give an alternative description of the interpreter.

The clear advantage is that such a description is shorter and simpler to read, and many implementation details vanish. The drawback of this approach is that this interpreter will be unbearably slow as long as it is run on top of CPython.

To get to a useful interpreter again, we need to translate our high-level description of Python to a lower level one. One rather straight-forward way is to do a whole program analysis of the PyPy interpreter and create a C source, again. There are many other ways but let us stick with this somewhat canonical approach.

10.2.1 Our Runtime Interpreter is "Restricted Python"

In order to make a C code generator feasible we restrict ourselves to a subset of the Python language, and we adhere to some rules which make translation to lower level languages more obvious.

Unlike source-to-source translations (like e.g. Starkiller) we start translation from live Python code objects which constitute our Python interpreter. When doing its work of interpreting bytecode our Python implementation must behave in a static way often referenced as "RPythonic".

However, when the PyPy interpreter is started as a Python program, it can use all of the Python language until it reaches interpretation runtime. That is, during initialization our program is free to use the full dynamism of Python, including dynamic code generation.

An example can be found in the current implementation which is quite elegant: For the definition of all the opcodes of the Python interpreter, the module `dis` is imported and used to initialize our bytecode interpreter. (See `__initclass__` in pyopcode.py). Thus we are saved from adding extra modules to PyPy. The import code is run at startup time, and we are allowed to use the CPython builtin import function.

After the startup code is finished, all resulting objects, functions, code blocks etc. must adhere to certain runtime restrictions which we describe further below. Here is some background why it is done like this: during translation, a whole program analysis ("type inference") is performed, which makes use of the restrictions defined in RPython. This enables the code generator to emit efficient machine level replacements for pure integer objects for instance.

10.3 RPython is not Strictly Defined

The list and exact details of the "RPython" restrictions are a somewhat evolving topic. In particular we have no formal language definition as we think it is more practical to discuss and evolve the set of restrictions while working on the whole program analysis.

10.3.1 Flow Restrictions

variables

variables should contain values of at most one type as described in Object restrictions at each control flow point, that means for example that joining control paths using the same variable to contain both a float and an int should be avoided. Mixing None (basically with the role of a null pointer) and *wrapped objects* and class instances is allowed.

constants

all module globals are considered constants.

control structures

all allowed but yield

range

`range` and `xrange` are identical. `range` does not necessarily create an array, only if the result is modified. It is allowed everywhere and completely implemented. The only visible difference to CPython is the inaccessability of the `xrange` fields start, stop and step.

definitions

run-time definition of classes or functions is not allowed.

generators

generators are not supported.

exceptions

- fully supported
- see below Exception rules for restrictions on exceptions raised by built-in operations

10.3.2 Object Restrictions

We are using

integer, float, string, boolean

avoid string methods and complex operations like slicing with a step

tuples

no variable-length tuples; use them to store or return pairs or n-tuples of values. Each combination of types for elements and length constitute a separate and not mixable type.

lists

lists are used as an allocated array. Lists are over-allocated, so list.append() is reasonably fast. Negative or out-of-bound indexes are only allowed for the most common operations, as follows:

- *indexing*: positive and negative indexes are allowed. Indexes are checked when requested by an IndexError exception clause.
- *slicing*: the slice start must be within bounds. The stop doesn't need to be, but it must not be smaller than the start. All negative indexes are disallowed, except for the [:-1] special case. No step.
- *other operators*: `+`, `+=`, `in`, `*`, `*=`, `==`, `!=` work as expected.
- *methods*: append, index, insert, extend, reverse, pop. The index used in pop() follows the same rules as for *indexing* above. The index used in insert() must be within bounds and must not be negative.

dicts

dicts with a unique key type only, provided it is hashable. String keys have been the only key types allowed for a while, but this was generalized. After some re-optimization, the implementation could safely decide that all string dict keys should be interned.

list comprehensions

may be used to create allocated, initialized arrays. After list over-allocation was introduced, there is no restriction any longer.

functions

- statically called functions may use defaults and a variable number of arguments (which may be passed as a list instead of a tuple, so write code that does not depend on it being a tuple).
- dynamic dispatch enforces use of very simple signatures, equal for all functions to be called in that context. At the moment this occurs in the opcode dispatch only.

builtin functions

A few builtin functions will be used while this set is not defined completely, yet. Some builtin functions are special forms:

len

- may be used with basic types that have a length. But len is a special form that is recognized by the compiler.
- If a certain structure is never touched by len, the compiler might save the length field from the underlying structure.

`int, float, ord, chr`... are available as simple conversion functions.

`int, float, str`... have a special meaning as a type inside of isinstance only.

classes

- methods and other class attributes do not change after startup
- inheritance is supported
- classes are first-class objects too

objects

wrapped objects are borrowed from the object space. Just like in CPython code that needs e.g. a dictionary can use a wrapped dict and the object space operations on it.

This layout makes the number of types to take care about quite limited.

10.3.3 Integer Types

While implementing the integer type we stumbled over the problem that integers are quite in flux in CPython right now. Starting on Python 2.2 integers mutate into longs on overflow. However shifting to the left truncates up to 2.3 but extends to longs as well in 2.4. By contrast we need a way to perform wrap-around machine-sized arithmetic by default while still being able to check for overflow when we need it explicitly. Moreover we need a consistent behavior before and after translation.

We use normal integers for signed arithmetic. It means that before translation we get longs in case of overflow, and after translation we get a silent wrap-around. Whenever we need more control, we use the following helpers:

ovfcheck()

This special function should only be used with a single arithmetic operation as its argument, e.g. `z = ovfcheck(x+y)`. Its intended meaning is to perform the given operation in overflow-checking mode.

At run-time, in Python, the ovfcheck() function itself checks the result and raises OverflowError if it is a `long`. But the code generators use ovfcheck() as a hint: they replace the whole `ovfcheck(x+y)` expression with a single overflow-checking addition in C.

ovfcheck_lshift()

ovfcheck_lshift(x, y) is a workaround for ovfcheck(x<<y), because the latter doesn't quite work in Python prior to 2.4, where the expression x<<y will never return a long if the input arguments are ints. There is a specific function ovfcheck_lshift() to use instead of some convoluted expression like `x*2**y` so that code generators can still recognize it as a single simple operation.

intmask()

This function is used for wrap-around arithmetic. It returns the lower bits of its argument, masking away anything that doesn't fit in a C "signed long int". Its purpose is, in Python, to convert from a Python `long` that resulted from a previous operation back to a Python `int`. The code generators ignore intmask() entirely, as they are doing wrap-around signed arithmetic all the time by default anyway. (We have no equivalent of the "int" versus "long int" distinction of C at the moment and assume "long ints" everywhere.)

r_uint

In a few cases (e.g. hash table manipulation) we need machine-sized unsigned arithmetic. For these cases there is the r_uint class which is a pure Python implementation of word-sized unsigned integers that silently wrap around. The purpose of this class (as opposed to helper functions as above) is consistent typing: both Python and the annotator will propagate r_uint instances in the program and interpret all the operations between them as unsigned. Instances of r_uint are special-cased by the code generators to use the appropriate low-level type and operations. Mixing of (signed) integers and r_uint in operations produces r_uint that means unsigned results. To convert back from r_uint to signed integers, use intmask().

10.3.4 Exception Rules

Exceptions are not generated for simple cases by default.:

```
#!/usr/bin/python

    lst = [1,2,3,4,5]
    item = lst[i]    # this code is not checked for out-of-bound access

    try:
        item = lst[i]
    except IndexError:
        # complain
```

Code with no exception handlers does not raise exceptions (after it has been translated, that is. When you run it on top of CPython, it may raise exceptions, of course). By supplying an exception handler you ask for error checking. Without that you assure the system that the operation cannot fail. This rule does not apply to *function calls*: any called function is assumed to be allowed to raise any exception.

For example:

```
x = 5.1
x = x + 1.2       # not checked for float overflow
try:
    x = x + 1.2
except OverflowError:
    # float result too big
```

But:

```
z = some_function(x, y)    # can raise any exception
try:
    z = some_other_function(x, y)
except IndexError:
    # only catches explicitly-raised IndexErrors in some_other_function()
    # other exceptions can be raised, too, and will not be caught here.
```

The ovfcheck() function described above follows the same rule: in case of overflow, it explicitly raise OverflowError, which can be caught anywhere.

Exceptions explicitly raised or re-raised will always be generated.

10.3.5 PyPy is Debuggable on Top of CPython

The advantage of PyPy is that it is runanble on standard CPython. That means, we can run all of PyPy with all exception handling enabled, so we might catch cases where we failed to adhere to our implicit assertions.

10.3.6 Wrapping Rules

PyPy is made of Python source code at two levels: On the one hand there is *application-level code* that looks like normal Python code, and that implements some functionalities as one would expect from Python code (e.g. one can give a pure Python implementation of some built-in functions like `zip()`). On the other hand there is *interpreter-level code* for the functionalities that must manipulate interpreter data and objects more directly (e.g. the main loop of the interpreter, and the various object spaces).

Application-level code doesn't see object spaces explicitly: it runs using an object space to support the objects it manipulates, but this is implicit. There is no need for particular conventions for application-level code. The sequel only is about interpreter-level code. (Ideally no application-level variable should be called `space` or `w_xxx` to avoid confusion.)

10.3.7 Naming Conventions

- `space`: the object space is only visible at interpreter-level code, where it is passed around by convention by the name `space`.
- `w_xxx`: any object seen by application-level code is an object explicitly managed by the object space. From the interpreter-level point of view, this is called a *wrapped* object. The `w_` prefix is used for any type of application-level object.
- `xxx_w`: an interpreter-level container for wrapped objects, for example a list or a dict containing wrapped objects. Not to be confused with a wrapped object that would be a list or a dict: these are normal wrapped objects, so they use the `w_` prefix.

10.3.8 Operations on `w_xxx`

The core bytecode interpreter considers wrapped objects as black boxes. It is not allowed to inspect them directly. The allowed operations are all implemented on the object space: they are called `space.xxx()`, where `xxx` is a standard operation name (`add`, `getattr`, `call`, `eq`...). The list of standard operations is found in the large table near the end of the module `pypy.interpreter.baseobjspace`. These operations take wrapped arguments and return a wrapped result (or sometimes just None).

Also note some helpers:

- `space.call_function(w_callable, ...)`: collects the given (already-wrapped) arguments, builds a wrapped tuple for them, and uses `space.call()` to perform the call.
- `space.call_method(w_object, 'method', ...)`: uses `space.getattr()` to get the method object, and then `space.call_function()` to invoke it.

10.3.9 Building `w_xxx` Objects

>From the bytecode interpreter, wrapped objects are usually built as the result of an object space operation. The ways to directly create a wrapped object are:

- `space.wrap(x)`: returns a wrapped object containing the value `x`. Only works if `x` is either a simple value (integer, float, string) or an instance of an internal bytecode interpreter class (Function, Code, Frame...).

- `space.newlist([w_x, w_y, w_z...])`: returns a wrapped list from a list of already-wrapped objects.
- `space.newtuple([w_x, w_y, w_z...])`: returns a wrapped tuple from a list of already-wrapped objects.
- `space.newdict([])`: returns a new, empty wrapped dictionary. (The argument list can contain tuples `(w_key, w_value)` but it seems that such a use is not common.)
- `space.newbool(x)`: returns `space.w_False` or `space.w_True` depending on the truth value of x.

There are a few less common constructors, described in the comments at the end of the module `pypy.interpreter.baseobjspace`.

10.3.10 Constant w_xxx Objects

The object space holds a number of predefined wrapped objects. The most common ones are `space.w_None` and `space.w_XxxError` for each exception class `XxxError` (for example `space.w_KeyError`, `space.w_IndexError`, etc.).

10.3.11 Inspecting and Unwrapping w_xxx Objects

The most delicate operation for the bytecode interpreter is to inspect a wrapped object, which must be done via the object space.

- `space.is_true(w_x)`: checks if the given wrapped object is considered to be `True` or `False`. You must never use the truth-value of w_x directly; doing so (e.g. writing `if w_x:`) will give you an error reminding you of the problem.
- `w_x == w_y` or `w_x is w_y`: DON'T DO THAT. The rationale for this rule is that there is no reason that two wrappers are related in any way even if they contain what looks like the same object at application-level. To check for equality, use `space.eq_w(w_x, w_y)` returning directly a interpreter-level bool (this is a short cut for `space.is_true(space.eq( w_x, w_y))`). To check for identity, use `space.is_w(w_x, w_y))`.
- `space.unpackiterable(w_x)`: this helper iterates w_x (using the operations `space.iter()` and `space.next()`) and collects the resulting wrapped objects in a list. Of course, in cases where iterating directly is better than collecting the elements in a list first, you should use `space.iter()` and `space.next()` directly.
- `space.unwrap(w_x)`: inverse of `space.wrap()`. Attention! Using `space.unwrap()` must be avoided whenever possible, i.e. only use this when you are well aware that you are cheating, in unit tests or bootstrapping code.
- `space.interpclass_w(w_x)`: If w_x is a wrapped instance of an interpreter class - for example Function, Frame, Cell, etc. - return it unwrapped. Otherwise return None.
- `space.int_w(w_x)`: If w_x is an application-level integer or long which can be converted without overflow to an integer, return an interpreter-level integer. Otherwise raise TypeError or OverflowError.
- `space.str_w(w_x)`: If w_x is an application-level string, return an interpreter-level string. Otherwise raise TypeError.

- `space.float_w(w_x)`: If w_x is an application-level float, integer or long, return interpreter-level float. Otherwise raise TypeError or OverflowError in case of very large longs.

Remember that you can usually obtain the information you want by invoking operations or methods on the wrapped objects; e.g. `space.call_method(w_dict, 'iterkeys')` returns a wrapped iterable that you can decode with `space.unpackiterable()`.

10.3.12 Application-level Exceptions

Interpreter-level code can use exceptions freely. However all application-level exceptions are represented as an `OperationError` at interpreter-level. In other words all exceptions that are potentially visible at application-level are internally an `OperationError`. This is the case for all errors reported by the object space operations (`space.add()` etc.).

To raise an application-level exception:

```
raise OperationError(space.w_XxxError, space.wrap("message"))
```

To catch a specific application-level exception:

```
try:
    ...
except OperationError, e:
    if not e.match(space, space.w_XxxError):
        raise
    ...
```

This construct catches all application-level exceptions, so we have to match it against the particular `w_XxxError` we are interested in and re-raise other exceptions. The exception instance e holds two attributes that you can inspect: `e.w_type` and `e.w_value`. Do not use `e.w_type` to match an exception, as this will miss exceptions that are instances of subclasses.

We are thinking about replacing `OperationError` with a family of common exception classes (e.g. `AppKeyError`, `AppIndexError`...) so that we can catch them more easily. The generic `AppError` would stand for all other application-level classes.

10.4 Modules in PyPy

Modules visible from application programs are imported from interpreter or application level files. PyPy reuses almost all Python modules of CPython's standard library, currently from version 2.4.1. We sometimes need to modify modules and - more often - regression tests because they rely on implementation details of CPython.

If we don't just modify an original CPython module but need to rewrite it from scratch we put it into *pypy/lib/* as a pure application level module.

When we need access to interpreter-level objects we put the module into *pypy/module*. Such modules use a mixed module mechanism which makes it convenient to use both interpreter- and applicationlevel parts for the implementation. Note that there is no extra facility for pure-interpreter level modules because we haven't needed it so far.

10.4.1 Determining the Location of a Module Implementation

You can interactively find out where a module comes from, here are examples for the possible locations:

```
>>>> import sys
>>>> sys.__file__
'/home/hpk/pypy-dist/pypy/module/sys/__init__.pyc'

>>>> import operator
>>>> operator.__file__
'/home/hpk/pypy-dist/pypy/lib/operator.py'

>>>> import types
t>>>> types.__file__
'/home/hpk/pypy-dist/lib-python/modified-2.4.1/types.py'

>>>> import os
faking <type 'posix.stat_result'>
faking <type 'posix.statvfs_result'>
>>>> os.__file__
'/home/hpk/pypy-dist/lib-python/2.4.1/os.py'
>>>>
```

10.4.2 Module Directories / Import Order

Here is the order in which PyPy looks up Python modules:

pypy/modules

mixed interpreter/app-level builtin modules, such as the `sys` and `__builtin__` module.

contents of PYTHONPATH

lookup application level modules in each of the `:` separated list of directories, specified in the `PYTHONPATH` environment variable.

pypy/lib/

contains pure Python reimplementation of modules.

lib-python/modified-2.4.1/

The files and tests that we have modified from the CPython library.

lib-python/2.4.1/

The unmodified CPython library. **Never ever checkin anything here**.

10.4.3 Modifying a CPython Library Module or Regression Test

Although PyPy is very compatible with CPython we sometimes need to change modules contained in our copy of the standard library. Often this is due to the fact that PyPy works with all new-style classes by default and CPython has a number of places where it relies on some classes being old-style.

If you want to change a module or test contained in `lib-python/2.4.1` then make sure that you copy the file to our `lib-python/modified-2.4.1` directory first. In subversion commandline terms this reads:

```
svn cp lib-python/2.4.1/somemodule.py lib-python/modified-2.4.1/
```

and subsequently you edit and commit `lib-python/modified-2.4.1/somemodule.py`. This copying operation is important because it keeps the original CPython tree clean and makes obvious what we had to change.

10.4.4 Implementing a Mixed Interpreter/Application Level Module

If a module needs to access PyPy's interpreter level it has to be implemented as a mixed module.

Mixed modules are directories in *pypy/module* with an *__init__.py* file containing specifications where each name in a module comes from. Only specified names will be exported to a Mixed Module's applevel namespace.

Sometimes it is neccessary to really write some functions in C (or whatever is the target language). See the external functions documentation for details.

10.4.5 Application Level Definitions

Application level specifications are found in the *appleveldefs* dictionary found in `__init__.py` files of directories in `pypy/module`. For example, in `pypy/module/__builtin__/__init__.py` you find the following entry specifying where `__builtin__.locals` comes from:

```
...
'locals'        : 'app_inspect.locals',
...
```

The `app_` prefix indicates that the submodule `app_inspect` is interpreted at application level and the wrapped function value for `locals` will be extracted accordingly.

10.4.6 Interpreter Level Definitions

Interpreter level specifications are found in the `interpleveldefs` dictionary found in the `__init__.py` files of directories in `pypy/module`. For example, in the `__init__.py` of the directory `pypy/module/__builtin__` the following entry specifies where `__builtin__.len` comes from:

```
...
'len'       : 'operation.len',
...
```

The `operation` submodule lives at interpreter level and `len` is expected to be exposable to application level. Here is the definition for `operation.len()`:

```
def len(space, w_obj):
    """len(object) -> integer

    Return the number of items of a sequence or mapping."""
    return space.len(w_obj)
```

Exposed interpreter level functions usually take a `space` argument and some wrapped values (see wrapping rules).

You can also use a convenient shortcut in `interpleveldefs` dictionaries: namely an expression in parentheses to specify an interpreter level expression directly (instead of pulling it indirectly from a file):

```
...
'None'          : '(space.w_None)',
'False'         : '(space.w_False)',
...
```

The interpreter level expression has a `space` binding when it is executed.

10.4.7 Testing Modules in pypy/lib

You can go to the *pypy/lib/test2* directory and invoke the testing tool ("py.test" or "python ../../test_all.py") to run tests against the pypy/lib hierarchy. Note that tests in *pypy/lib/test2* are allowed and encouraged to let their tests run at interpreter level although *pypy/lib/* modules eventually live at PyPy's application level. This allows us to test our Python-coded reimplementations against CPython quickly.

10.4.8 Testing Modules in pypy/module

Simply change to `pypy/module` or to a subdirectory and run the tests as usual.

10.4.9 Testing Modules in lib-python

In order to let CPython's regression tests run against PyPy you can switch to the *lib-python/* directory and run the testing tool in order to start compliance tests. (XXX check windows compatibility for producing test reports).

10.5 Naming Conventions and Directory Layout

10.5.1 Directory and File Naming

- directories/modules/namespaces are always **lowercase**
- never use plural names in directory and file names

- `__init__.py` is usually empty except for those in `pypy/objspace/*` and, as described above, `pypy/module/*/__init__.py`.
- don't use more than 4 directory nesting levels
- keep filenames concise and completion-friendly.

10.5.2 Naming of Python Objects

- class names are **CamelCase**
- functions/methods are lowercase and _ separated
- objectspace classes are spelled `XyzObjSpace`. e.g.
  - StdObjSpace
  - FlowObjSpace
- at interpreter level and in ObjSpace all boxed values have a leading `w_` to indicate "wrapped values". This includes w_self. Don't use `w_` in application level Python only code.

10.5.3 Committing & Branching to the Repository

- write good log messages because several people are reading the diffs.
- if you add (text/py) files to the repository then please run pypy/tool/fixeol in that directory. It ensures that the property 'svn:eol-style' is set to native which allows checkin/checkout in native line-ending format.
- branching (aka "svn copy") of source code should usually happen at `svn/pypy/dist` level in order to have a full self-contained pypy checkout for each branch. For branching a `try1` branch you would for example do:

  ```
  svn cp http://codespeak.net/svn/pypy/dist \
         http://codespeak.net/svn/pypy/branch/try1
  ```

  This allows to checkout the `try1` branch and receive a self-contained working-copy for the branch. Note that branching/copying is a cheap operation with subversion, as it takes constant time irrespective of the size of the tree.
- To learn more about how to use subversion read this document.

10.6 Using the Development Bug/Feature Tracker

We have a development tracker based on Richard Jones' roundup application. You can file bugs, feature requests or see what's going on for the next milestone, both from an E-Mail and from a web interface.

10.7 Use your Codespeak Login or Register

If you already committed to the PyPy source code chances would be that you can simply use your codespeak login that you use for subversion or for shell access.

If you are not a commiter then you can still register with the tracker easily.

10.8 Modifying Issues from svn Commit Messages

If you are committing something related to an issue in the development tracker you can correlate your login message to a tracker item by following these rules:

- put the content of `issueN STATUS` on a single new line
- *N* must be an existing issue number from the development tracker.
- STATUS is one of:

```
unread
chatting
in-progress
testing
duplicate
resolved
```

10.9 Testing in PyPy

Our tests are based on the new py.test tool which lets you write unittests without boilerplate. All tests of modules in a directory usually reside in a subdirectory **test**. There are basically two types of unit tests:

- **Interpreter Level tests**. They run at the same level as PyPy's interpreter.
- **Application Level tests**. They run at application level which means that they look like straight Python code but they are interpreted by PyPy.

Both types of tests need an objectspace they can run with (the interpreter dispatches operations on objects to an objectspace). If you run a test you can usually give the '-o' switch to select an object space. E.g. '-o thunk' will select the thunk object space. The default is the Standard Object Space which aims to implement unmodified Python semantics.

10.9.1 Interpreter Level Tests

You can write test functions and methods like this:

```
def test_something(space):
    # use space ...

class TestSomething:
    def test_some(self):
        # use 'self.space' here
```

Note that the prefix *test* for test functions and *Test* for test classes is mandatory. In both cases you can import Python modules at module global level and use plain 'assert' statements thanks to the usage of the py.test tool.

10.9.2 Application Level Tests

For testing the conformance and well-behavedness of PyPy it is often sufficient to write "normal" application-level Python code that doesn't need to be aware of any particular coding style or restrictions. If we have the choice we often use application level tests which usually look like this:

```
def app_test_something():
    # application level test code

class AppTestSomething:
    def test_this(self):
        # application level test code
```

These application level test functions will run on top of PyPy, i.e. they have no access to interpreter details. You cannot use imported modules from global level because they are imported at interpreter-level while you test code runs at application level. If you need to use modules you have to import them within the test function.

Another possibility to pass in data into the AppTest is to use the `setup_class` method of the AppTest. All wrapped objects that are attached to the class there and start with `w_` can be accessed via self (but without the `w_`) in the actual test method. An example:

```
from pypy.objspace.std import StdObjSpace

class AppTestErrno:
    def setup_class(cls):
        cls.space = StdObjSpace()
        cls.w_d = cls.space.wrap({"a": 1, "b", 2})

    def test_dict(self):
        assert self.d["a"] == 1
        assert self.d["b"] == 2
```

10.9.3 Command Line Tool test_all

You can run almost all of PyPy's tests by invoking:

```
python test_all.py
```

which is a synonym for the general py.test utility located in the `pypy` directory. For switches to modify test execution pass the `-h` option.

10.9.4 Test Conventions

- adding features requires adding appropriate tests. (It often even makes sense to write the tests first so that you are sure that they actually can fail.)
- All over the pypy source code there are test/ directories which contain unittests. Such scripts can usually be executed directly or are collectively run by pypy/test_all.py
- each test directory needs a copy of pypy/tool/autopath.py which upon import will make sure that sys.path contains the directory where 'pypy' is in.

10.10 Changing Documentation and Website

10.10.1 Documentation Files in your Local Checkout

Most of the PyPy's documentation and website is kept in *pypy/doc* directory. You can simply edit or add '.txt' files which contain ReST-markuped files. Here is a ReST quickstart but you can also just look at the existing documentation and see how things work.

10.10.2 Automatically Test Documentation Changes

We automatically check referential integrity and ReST-conformance. In order to run the tests you need docutils installed. Then go to the local checkout of the documentation directory and run the tests:

```
cd .../pypy/documentation
python ../test_all.py
```

If you see no failures chances are high that your modifications at least don't produce ReST-errors or wrong local references. A side effect of running

the tests is that you have *.html* files in the documentation directory which you can point your browser to!

Additionally if you also want to check for remote references inside the documentation issue:

```
python ../test_all.py --checkremote
```

which will check that remote URLs are reachable.

11 Glossary of Abbreviations

The following abbreviations may be used within this document:

11.1 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.
docutils	The Python documentation utilities.
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.
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.

12 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