PyPy - The new Python Implemention on the Block

1 Mission statement

PyPy is an implementation of the Python[1] 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.

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"[2]).

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[3] that should make PyPy run Python programs faster than CPython, and extensions, following the ideas of Stackless[4] and others, that will increase the expressive power available to python programmers.

3 PyPy - Meta Goals

PyPy is not only about writing another Python interpreter. Traditionally, interpreters are written in some target platform language like C/Posix, Java or C#. Each such interpreter provides a "mapping" from application source code to the target environment. One of the goals of the "all-encompassing" environments, like the .NET framework and to some extent the Java virtual machine, is to provide standardized and higher level functionalities to language implementors. This reduces the burden of having to write and maintain many interpreters or compilers.

PyPy is experimenting with a different approach. We are not writing a Python interpreter for a specific target platform. We have written a Python interpreter in Python, without many references to low-level details. (Because of the nature of Python, this is already a complicated task, although not as much as writing it - say - in C.) Then we use this as a "language specification" and manipulate it to produce the more traditional interpreters that we want. In the above sense, we are generating the concrete "mappings" of Python into lower-level target platforms.

So far (fall 2005), we have already succeeded in turning this "language specification" into reasonably efficient C-level code that performs basically the same job as CPython. Memory management is inserted during this "translation" process. It can be configured to use reference counting or not; thus we have already achieved two very different "mappings" of application Python code over C/Posix. We have successfully translated our Python interpreter to the LLVM[5] target as well, and we are

[1] http://www.python.org/doc/current/ref/ref.html
[2] http://python.org
[3] http://psyco.sourceforge.net
[4] http://stackless.com

working on targeting higher-level environments like Java and Squeak.

In some senses, PyPy project's central component is not its interpreter implementation, but its configurable translator. We think it provides a good way to avoid writing `n * m * o` interpreters for `n` dynamic languages and `m` platforms with `o` crucial design decisions. PyPy aims at having any one of these parameters changeable independently from each other:

- we can modify or replace the language we interpret and just regenerate a concrete interpreter for each target;
- we can write new translator back-ends to target new platforms;
- we can tweak the translation process to produce low-level code based on different models and tradeoffs.

By contrast, a standardized target environment – say .NET – enforces `m=1` as far as it is concerned. This helps making `o` a bit smaller by providing a higher-level base to build upon. Still, we believe that enforcing the use of one common environment is not necessary. PyPy's goal is to give weight to this claim - at least as far as language implementation is concerned - showing an approach to the `n * m * o` problem that does not rely on standardization.

This is the "meta-goal"; a more concrete goal worth mentioning at this point is that language specifications can be used to generate cool stuff in addition to traditional interpreters – e.g. Just-In-Time Compilers.

4 Higher level picture

As you would expect from a project implemented using ideas from the world of Extreme Programming[6], the architecture of PyPy has evolved over time and continues to evolve. Nevertheless, the high level architecture is now clear. There are two independent basic subsystems: the Standard Interpreter and the Translation Process.

4.1 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.

The *bytecode interpreter* is the part that interprets the compact bytecode format produced from user Python sources by a preprocessing phase, the *bytecode compiler*. The bytecode compiler itself is implemented as a chain of flexible passes (tokenizer, lexer, parser, abstract syntax tree builder, bytecode generator). The bytecode interpreter then does its work by delegating all actual manipulation of user objects to the *object space*. The latter can be thought of as the library of built-in types. It defines the implementation of the user objects, like integers and lists, as well as the operations between them, like addition or truth-value-testing.

This division between bytecode interpreter and object space is very important, as it gives a lot of flexibility. It is possible to use different object spaces to get different behaviours of the Python objects. Using a special object space is also an important technique for our translation process.

4.2 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, operations that fit the model of the target platform;

[5] http://llvm.cs.uiuc.edu/
[6] http://www.extremeprogramming.com/

- the *code generator* which translates the resulting flow graph into another language, currently C, LLVM, Javascript (experimental).

A more complete description of the phases of this process is out of the scope of the present introduction. We will only give a short overview in the sequel.

5 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 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 low-level values and data structures.

The actual low-level code is emitted by "visiting" the type-annotated flow graph after the typer introduced low-level operations. Currently we have a C-producing backend, and an LLVM-producing backend. The former also accepts non-annotated or partially-annotated graphs, which allow us to test it on a larger class of programs than what the Annotator can (or ever will) fully process. The complete translation process is described in more detail in the documentation section on the PyPy homepage[7].

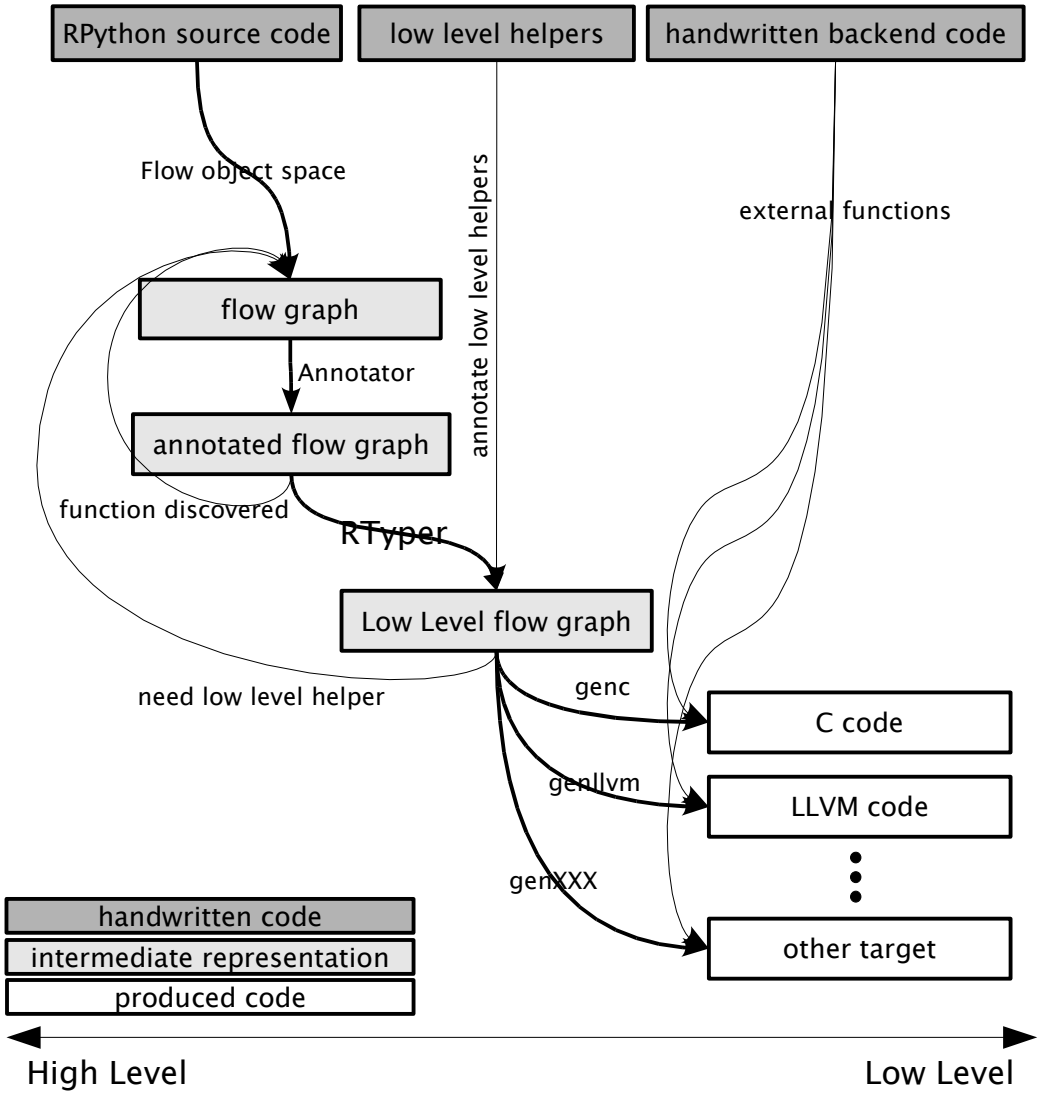
6 Status of the implementation (Oct 2005)

With the pypy-0.8.0 release we have integrated our AST compiler with the rest of PyPy. The compiler gets translated with the rest to a static self-contained version of our standard interpreter. Like with 0.7.0 this version is very compliant[8] to CPython 2.4.1 but you cannot run many existing programs on it yet because we are still missing a number of C-modules like socket or support for process creation.

The self-contained PyPy version (single-threaded and using the Boehm-Demers-Weiser garbage collector[9]) now runs around 10-20 times slower than CPython, i.e. around 10 times faster than 0.7.0. This is the result of optimizing, adding short cuts for some common paths in our interpreter and adding relatively straightforward optimization transforms to our tool chain, like inlining paired with simple escape analysis to remove unnecessary heap allocations. We still have some way to go, and we still expect most of our speed will come from our Just-In-Time compiler work, which we have barely started at the moment.

With the 0.8.0 release the "thunk" object space

[7] http://codespeak.net/pypy



overview over the translation process

can also be translated, obtaining a self-contained version of PyPy with its features (and some speed degradation), show-casing at a small scale how our whole tool-chain supports flexibility from the interpreter written in Python to the resulting self-contained executable.

Our rather complete and Python 2.4-compliant interpreter consists of about 30'000-50'000 lines of code (depending on the way you count code borrowed and adapted from other sources), with another 14'000 lines of unit tests. If we include the tools, the parts related to code analysis and generation, and the standard library modules ported from C, PyPy is now 138'000 lines of code and 32'000 lines of tests. Refer to the statistics web page[10] for more detailed information.

7 Future work and foreseen possibilities

In 2006, the PyPy project aims to translate the standard Python Interpreter to a JIT-compiler and also to support massive parallelism aka micro-threads within the language. These are not trivial tasks especially if we want to retain and improve the modularity and flexibility aspects of our implementation - like giving an independent choice of memory or threading models for translation. Moreover it is likely that our javascript and other higher level backends (in contrast to our current low-level ones) will continue to evolve.

Apart from optimization-related translation choices PyPy is to enable new possibilities regarding persistence, security and distribution issues. We intend to experiment with ortoghonal persistence for Python objects, i.e. one that doesn't require application objects to behave in a particular manner. Security wise we will look at sandboxing or capabilities based schemes. For distribution we already experimented with allowing transparent migration of objects between processes with the help of the existing (and translateable) Thunk Object Space. In general, according experiments are much easier to conduct with PyPy and should provide a resulting standalone executable in shorter time.

[8] http://www.hpl.hp.com/personal/Hans_Boehm/gc/
[9] http://codespeak.net/~hpk/pypy-testresult/
[10] http://codespeak.net/~hpk/pypy-stat/