IST FP6-004779

PYPY

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

STREP

IST Priority 2

D14.4 PyPy-1.0 Milestone report

Due date of deliverable: March 2007

Actual Submission date: May 2nd, 2007

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

Lead Contractor of this WP: Change Maker

Authors: Holger Krekel (merlinux), Beatrice Düring (Change Maker), Samuele Pedroni (Open End), Lene Wagner (merlinux)

Revision: Final

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

Dissemination Level: PU (Public)

PyPy D14.4: PyPy-1.0 Milestone Report

Revision History

Date	Name	Reason of Change
2007-02-25	Beatrice Düring	Disposition and notes for all sections
2007-03-29	Holger Krekel	Refined outline, Exec Summary Draft
2007-03-29	Samuele Pedroni	Added discussion of our JIT results
2007-03-30	Beatrice Düring	Draft of non-technical section, references
2007-03-30	Samuele Pedroni	Refactored structure after internal discussion
2007-03-31	Holger Krekel	New Exec Summary Draft, further refactoring
2007-04-02	Lene Wagner	Refactoring non-technical sections/content
2007-04-03	Beatrice Düring	Refinements to conclusion and appendix
2007-04-03	Holger Krekel	Abstract/Conclusion fixes after internal review
2007-04-05	Carl Friedrich Bolz	Publish Interim version on the Web Page
2007-05-01	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

The public 1.0 milestone release validates PyPy's as an interesting open-source platform for implementing dynamic languages. Its *translation framework* can analyse *language interpreters* written in a high level language, and transform them into efficient lower level versions. PyPy-1.0 includes a major and novel research result: PyPy can *generate a Just-In-Time Compiler* directly from such an interpreter, independently of the language being interpreted.

The main target of the framework continues to be a compliant and very flexible Python Interpreter. Its architecture has been validated by adding security, orthogonal persistence and distribution features, and extensions for logic, constraint and aspect oriented programming. The 1.0 milestone not only integrates all the according work but also includes many community contributions that support PyPy's vision to become a language implementation platform not relying on standardization, but on its powerful metaprogramming translation approach.

Purpose, Scope and Related Documents

This document is intended to be read by developers, researchers and other other parties interested in language interpreter implementations. It provides an high level overview on results of the PyPy project with its 1.0 milestone release. It is a good entry point for understanding PyPy's key approaches both on a technical and a community/development level, and to find out about available more detailed reports and documentation.

Related Reports:

- D02.1 Configuration and Maintenance of Development Tools and website [D02.1]
- D02.3 Automated Testing and Development support library [D02.3]
- D03.1 Synchronization with Standard Python and the Extension Compiler [D03.1]
- D07.1 Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects [D07.1]
- D08.2 JIT Compiler Architecture [D08.2]

- D09.1 Constraint Satisfaction and Inference Engine, Logic Programming in Python [D09.1]
- D10.1 Aspect-Oriented, Design-by-Contract Programming and RPython static checking [D10.1]
- D11.1 A Case Study On Using PyPy For Embedded Devices [D11.1]
- D12.1 High-Level Backends and Interpreter Feature Prototypes [D12.1]
- D13.1 Integration and Configuration [D13.1]
- D14.3 Milestone 2 report [D14.3]
- D14.5 Development Process [D14.5]

Contents

1 Executive Summary

With its 1.0 milestone release [PYPY-1.0] PyPy evolves as an interesting approach for implementing dynamic languages. Relying on metaprogramming techniques rather than standardization, PyPy offers flexibility and speed: language interpreters may be implemented on a very high level of abstraction and then successively analysed and transformed into efficient versions for hardware and virtual platforms. In fact, PyPy 1.0 can now *generate* Just-In-Time compilers that - independently of the interpreter specification being analysed - can increase runtime speed of interpreted high level algorithmic code close to "C" level.

PyPy 1.0 also is the result of a sprint-driven and dispersed development model combining open source culture and language research within a contractual EU funding context. Based on a publicly visible and open development structure, a mix of entry-level documentation, tutorial sessions and mentoring of selected proposals helped newcomers to enter the project and enrich it with ideas and contributions. One direct consequence of community contributions is the multitude of practically usable PyPy backends (C, LLVM, .NET, JavaScript, JVM) and the beginnings of a JavaScript interpreter, overall pushing PyPy towards an open source platform for implementing dynamic languages.

New research projects are likely to emerge that use and advance specific areas of PyPy. A vital community shares an interest in making PyPy's Python interpreter more mainstream and usable, and to experiment with its flexibility. The possibility of implementing interpreters for languages other than Python, should receive more attention after the PyPy 1.0 milestone. More development directions include using RPython as a base for web or other special purpose applications. To keep the conceptual integrity of the PyPy project as a whole, a group of trusted contributors is to take responsibility and also aims at getting funding for PyPy's sprint driven development model which should continue to bring key persons and new developers together.

2 Technical Summary of the PyPy 1.0 Milestone

The last phase of the PyPy EU project focused on proving the real and effective advantages of PyPy's architecture and approach. We developed prototypes for usually challenging features and worked to bring to fruition our original ideas and optimization efforts, up to a new way to deliver dynamic optimizations to very high level languages. Significant efforts went into integrating our various developments.

PyPy's technical claim is that interpreters for dynamic languages can be implemented on a high level of abstraction, yet be brought to run at least as efficiently as today's common lower level implementations.

Our PyPy 1.0 milestone release gives weight to this claim. It validates high level architectural aspects of both our Python interpreter and our translation approach. PyPy 1.0 integrates all results of our development and research efforts during the last four years [PYPY-1.0]. PyPy contains both:

- A flexible Python interpreter written in *RPython*, and
- An advanced *translation framework* able to translate such interpreters into low-level versions.

PyPy's approach to language implementation uses metaprogramming techniques: *language interpreters* are implemented in an object language, RPython - a restricted subset of Python amenable to type-inference. The *translation* metaprograms analyse such interpreters, perform whole-program type inference and successively transform and enrich internal representations in order to eventually generate platform specific versions of the interpreter. In the process any necessary or custom low-level details (memory management, threading models, optimizations) are interweaved and efficiency is recovered to a large extent. The translation metaprograms itself are written using full Python with all its expressiveness and can be applied to any interpreter written in RPython.

An important focus of development during the last phase was to validate and integrate results of all areas of PyPy development [D13.1], particularly implementing systematic and complete configuration facilities.

2.1 Static and Dynamic Optimizations

Being able to write an interpreter in a high-level language and to statically translate it to relatively efficient lower level versions addresses maintenance and flexibility issues, but leaves the performance question partly unanswered. Dynamic languages need dynamic Just-In-Time optimizations for best-of-breed performance.

PyPy 1.0 includes the final piece of our original vision: a novel practical approach to generate Just-In-Time Compilers [D08.2], an area of language development that traditionally interweaves lower level implementation and language specification details. PyPy, by contrast, introduces JIT Compilers as another step in its translation transformations: it analyses and transforms any interpreter implementation and - given a few hints - can generate a low level version of the interpreter that includes Just-In-Time specializing code generation. With PyPy 1.0 this already provides JIT-typical speed benefits for algorithmic examples.

A *JIT compiler generator* renders the issue of performance mostly orthogonal to the evolution and maintenance of the language interpreter. This is radically different from simply writing such a dynamic compiler in a high-level language. Such a compiler needs itself to encode the semantics of the language and this kind of encoding is usually hard to understand and maintain, the harder the more rich and dynamic the language in question. Changes to the language needs to be reflected into the compiler. PyPy's approach lets the implementation be just an interpreter, and the JIT generation technology is applicable to the interpreter of today as to that of tomorrow and to other interpreters for other languages as well.

Even independently from JIT technology, the result of applying optimizations both to our Python interpreter [D06.1] and during translation [D07.1] have been quite positive: our compliant and flexible generated Python interpreter runs benchmarks at about half of CPython's speed, the reference Implementation of Python that has been manually optimized over the course of 15 years. With more refined garbage collection techniques we believe that we can get very close to CPython speed, i.e. not even considering JIT technologies which will definitely provide a significant edge. Our static and Interpreter optimizations were implemented on a high level of abstraction and independently from the dynamic JIT related efforts; PyPy clearly provides a good base for localized experimentation and optimizations.

2.2 Python Interpreter Validation and Extensions

Independently from optimizations and transformations within our translation framework, we validated the architecture of our Python interpreter specification. The 1.0 milestone includes prototypes and successful experiments for advanced feature that usually require invasive changes. We implemented a prototype with information flow control security features and added a general facility for transparent proxying of builtin objects, which we use to develop prototypes and proofs of concept for transparent distribution of objects across networks and orthogonal persistence [D12.1].

We also show that traditional Aspect Oriented Programming Techniques can be successfully integrated with our source parsing and compiling modules [D10.1]. Logic and Constraint Programming extensions have been integrated with our Python interpreter [D09.1], making use of "Stackless" and particular translation transformations [D07.1].

Overall the success of implementing such extensions and features validates our Python interpreter architecture and the effectiveness of our metaprogramming translation and transformation approach and research [VMC].

However, with its PyPy 1.0 milestone release PyPy's Python interpreter is still not generally usable. The main blocking factor is the lack of extension modules. We have developed a very interesting dual-use tool for implementing extension modules for PyPy and CPython from a single source code [D03.1]. There remain some deficiencies and more efforts are needed to improve usability. The project consciously decided early to first aim at realizing the full vision of PyPy's original claim, targeting the research oriented results before putting engineering and refactoring efforts into better extension module support. Indeed PyPy 1.0 has some of the rough edges that can be expected from the offspring of a research effort like PyPy.

3 Community and development process

The 1.0 milestone reflects a growing community around PyPy, supported and fostered by activities to ease entry into the project, such as improving documentation and tutorials [D14.2], giving lively insight into project backgrounds through video documentation [VIDEO], and, most importantly, following a test-driven development approach and the organisation of sprints, arising as a hybrid methodology that has received major scientific and public attention [D14.5].

PyPy is built on thousands of automated tests in all its coding areas, being run regularly to ensure the quality and consistency of the codebase. This test driven approach lowers the entry barrier for newcomers as they can experiment with changes and more easily ensure that they don't disrupt other functionality. The tests are driven by the `py.test` tool which has been separately released [D02.3] and presented at multiple conferences, and is increasingly used by many other projects and organisations.

3.1 Community contributions and commercial perspectives

Community contributions have greatly influenced PyPy during the last phase of the EU project, resulting in new features and directions, not foreseen at the start of the EU project. For example, as of PyPy 1.0 RPython programs can be translated to JavaScript as part of a practically interesting approach to the development of AJAX[1] web applications. We showcase such usage through our online demonstration server [PLAY1], which also allows a user to interact with the various generated PyPy interpreters and provides entry points into our large online documentation base [PYPYDOC]. Another important community contribution is the beginning of a JavaScript Interpreter that may evolve as the next major dynamic language interpreter making use of PyPy's translation framework. Also originating from a community proposal, `py.test` received new ad-hoc methods to distribute tests across many computers, which helped to speed up the development process particularly for the time-consuming dynamic optimization tests.

Commercial interest focused around PyPy's interpreter implementation language, RPython (see [D11.1] for a case study on using it for embedded systems). RPython can be seen as a "better Java" in that one can implement programs on a relatively high level, speeding up and simplifying the development process. Afterwards the *translation framework* can turn such programs into efficient low level versions, and with PyPy 1.0 there is the new possibility to make use of dynamic optimization (JIT) techniques, increasing eventual run-time speed. However, the development team originally designed RPython "just" as the implementation language for its Python interpreter. During the EU project period it was not feasible to focus on improving and enriching RPython for other purposes but this may change also with respect to future commercial usages.

Effectively, PyPy's Python interpreter implementation will continue to put the highest demands on the *translation framework*. It can be expected to foster commercial usage of its advanced features once it becomes a viable substitute for other major Python implementations. This in turn requires a self-sustaining and dedicated community and thus dissemination efforts during the last phase of the EU period focused much on the sustainability of the open source community around PyPy and its main target, the Python Interpreter.

3.2 Sprint-Driven Development and Sprint funding

During the last phase of the EU project, sprints (week long co-located coding sessions arranged every 6-8th week of the project) continued to be a key driving factor for the project [D14.5]. However, during that period, the project experienced a growing need to balance contractual obligations with a community wanting to follow its own ideas and priorities. It would become more difficult to combine the mentoring approach with the complex and time-critical contractual tasks within one sprint.

The project reacted with refinements to the sprinting approach, pre-targeting the sprints either at reaching important internal milestones or at introducing and mentoring newcomers. Sprints also increasingly served to bring

[1] AJAX, or "Asynchronous JavaScript and XML", is a relatively new web development technique for creating interactive web applications.

the management and development team together and to advance report and administrative work during the last months of the EU project.

Inspired by Google's Summer of Code [CODE], where core developers in PyPy participated 2005 and 2006 as mentors, PyPy launched its Summer of PyPy [SOP] initiative during the last phase of the project. A call for proposals was issued and five Summer of PyPy participants were approved who attended one or more sprints and in all cases successfully contributed to PyPy.

The offer to have people participate in PyPy sprints with their own idea and mentoring and supporting their efforts was an attractive incentive and increased the group of core developers. The work on the proposals lead to and triggered the eventual implementation of the already mentioned JavaScript interpreter, a Javascript/AJAX RPython mapping, a CLI backend, a JVM backend and several other contributions [D12.1]. Newcomers not only entered through PyPy sprints, but also through more traditional means of open source development: IRC channels and mailing lists.

The interest in the unique practice of development used in PyPy peaked during the last phase of the project. PyPy published experience reports at the two main Agile Community conferences in Europe and USA [TROUBLE], [SPRINT].

Interest in the practice and the community effects of its implementation within PyPy also spread to other EU-projects and resulted in various publications and collaborations, most notably the IEEE Software Magazine article "Sprinting towards Open Source Development" [IEEE] and a submitted, not yet accepted chapter for the book *Successful OSS Project Design and Implementation: Requirements, Tools, Social Designs, Reward Structures and Coordination Methods* [FLOSS]

Parts of the research community studying the F/OSS phenomenon also took a closer look at the sprint practice as implemented in PyPy [LEARN].

4 Conclusion and Outlook

Interesting areas for future research and engineering include:

- refining and advancing dynamic optimization capabilities
- implementations of non-Python interpreters (JavaScript, Prolog, ...) and also custom ones for games or embedded devices
- optimizing the high level backends (particularly CLI and JVM)
- incorporating advanced garbage collection techniques
- more Python interpreter and static optimizations
- using the flexibility of the Python interpreter to provide new middle-ware features for application development

To bring the PyPy project to its full potential it will be important to involve more non-Python developers interested in producing new language interpreters by utilizing the *translation framework*. During the last project phase, the PyPy team took care to open up to non-Python developers and received some encouraging public comments:

> "My point was that doing the types of things the PyPy team has done is HARD. So hard that even with great funding, lots of experience, and quite a bit of time, they still aren't where they might want to be yet.

> And therefore, for some of these other teams, it might make a lot more sense to harness the work the PyPy team has already done instead of doing it all on their own.
>
> It makes sense from a developers' standpoint, and it really makes sense from a users' standpoint, because dynamic languages need to get to the point one day where they are all using the same libraries. It is wasteful for Python, Ruby, and Perl to all be doing their own thing with libraries." [REDDIT1]

In fact, PyPy's *translation framework* should be a fertile ground to implement more dynamic languages and experiment with interoperability between them. It possibly provides a richer and more expressive setting than approaches building on top of standardized virtual machines like the JVM or the CLR (see also the *high level goals* of the [ARCH] architecture overview document).

PyPy's own *Python interpreter* has to surpass one engineering obstacle towards mainstream adoption: refining and refactoring its extension module support to provide a convenient and complete model for implementing such modules and gluing external libraries. However, with PyPy's increasing visibility as a flexible and fast platform and its compliant and feature rich Python interpreter, it should not be hard to gather community focus around removing this last obstacle.

Commercial interest continues to focus on using RPython for special purposes. Once the Python interpreter becomes more usable for mainstream applications, interest in utilizing its flexibility and speed features can be expected as well. To keep a balance between the interests of community, commercial and core developers, organising sprints will remain as a key approach to keeping conceptual integrity and bringing together PyPy developers, newcomers and interested parties.

5 Appendix: Dissemination facts from Phase 3 of the PyPy project

In this appendix we summarize key dissemination results that were achieved during the last phase of the project, these results are tied to activities such as sprints, conferences, publications, workshops and collaboration with external parties.

5.1 Sprints

Six sprints were arranged during the last phase of the project.

- Düsseldorf (June 2006), 12 participants
- Europython/Geneva (July 2006) 24 participants
- Limerick (Aug 2006) 7 participants
- Düsseldorf (October 2006), 14 participants
- Leysin (January 2007) 17 participants
- Hildesheim (February 2007) 19 participants.

5.2 Collaborations with external parties and communities

Besides the ongoing community (Python and PyPy) dialogue, which resulted in vital contributions to the PyPy framework, other collaborations worth mentioning are:

- Software Technologies Concertation meeting,Brussels, Belgium (September 2006). Dissemination conference for all projects within FP6 (software engineering/complexity, service engineering and open source). PyPy presented project objectives and results and participated in the open source session.
- PyPy continued to collaborate loosely with the Calibre [CAL] project during the last phase of the project. The main collaboration event was to participate in the Calibration internal workshop at Phillips Medical System where PyPy presented a talk on sprint-driven development in F/OSS projects.
- PyPy met with Andrea Glorioso from Institute Politecnico and the MUSIC project at the above mentioned Dissemination Conference. Both parties realized they had similar experiences worth documenting regarding F/OSS communities and public funding. The result was an accepted proposal and a full submitted chapter for call launched by the Business School of the University of Montpellier for the book ”Successful OSS Project Design and Implementation: Requirements, Tools, Social Designs, Reward Structures and Coordination Methods“ (to be published in Winter 2007). The chapter is titled ”E(U)volving FLOSS communities - How FLOSS projects can stop worrying and learn to love public funding“.
- The University of Limerick hosted a PyPy sprint in August 2006. During the sprint a workshop was arranged for interested researchers, mainly from the socGDS team which filmed the sprint and later published a paper [LEARN].
- We arranged a workshop with IBM in Zuerich (Switzerland) in January 2007 and discussed PyPy’s security prototype [D12.1] with Prof. Michael Franz and Dr. Matthias Schunter. It turned out that our effort in prototyping ”Tainted objects“ basically implements the state-of-the-art for a practical dynamic language. In addition, Mr. Franz, also a well-known researcher on JIT-Compiler topics, acknowledged and appreciated PyPy’s intermediate results in that area as well.
- PyPy was also invited to give a technical presentation for the Software Solutions Group at Intel Lab Hillsboro OR, where an extended version of the DLS talk was presented (also touching JIT generation) was given.

5.3 Conferences (Agile, Python, Object-Oriented Languages) and publications

- XP 2006, Oulu, Finland (June 2006): main agile european conference. PyPy published and experience report and presented at the conference. ”Sprint Driven Development, Agile Methodologies in a Distributed Open Source Project (PyPy)“
- Agile 2006, Minneapolis, Minnesota, USA (July 2006): main american agile conference. PyPy published an experience report and presented at the conference: ”Trouble in Paradise: the Open Source project PyPy, EU-funding and Agile practices“
- EuroPython 2006, CERN, Geneva, Switzerland (July 2006): main European Python conference in Europe. PyPy members participated and presented four different talks: ”An Introduction to PyPy“, ”PyPy architecture session“, ”What can PyPy do for you¿‘ and ”Kill-1: refactoring the PyPy process“.
- OOPSLA, Portland,Oregon, USA (October 2006): the main conference for object-oriented programming, systems, languages and applications. PyPy members published a paper and participated in the Dynamic Language Symposium, a companion conference to OOPSLA. ”PyPy’s approach to virtual machine construction“
- PyCon, Dallas, Texas, USA (February 2007): the largest Python conference in the Python community. PyPy participated and presented as talk: ”Towards and beyond PyPy 1.0“
- PyPy members presented two talks at Warzaw University, Poland (December 2006 and March 2007). ”PyPy fireworks“ and ”Massive parallelism“
- ”PyPy - the ’Babelfish’ of programming“, IST Results [BABEL]

References

[CODE] *Summer of Code*, http://code.google.com/soc/

[PLAY1] *Play1 PyPy Online Demo Server*, http://play1.pypy.org

[SOP] *Summer of PyPy*, http://codespeak.net/pypy/dist/pypy/doc/summer-of-pypy.html

[CAL] *Coordination actionCalibre project*, http://www.calibre.ie

[PYPY-1.0] *PyPy 1.0 release announcement*, http://codespeak.net/pypy/dist/pypy/doc/release-1.0.0.html

[VIDEO] *PyPy video documentation*, http://codespeak.net/pypy/dist/pypy/doc/video-index.html

[TROUBLE] *Trouble in Paradise: the Open Source project PyPy, EU-funding and Agile practices*, Agile 2006, Minneapolis

[SPRINT] *Sprint Driven Development, Agile Methodologies in a Distributed Open Source Project (PyPy)*, XP 2006, Oulu, Finland

[IEEE] *Sprinting towards Open Source Development*, Greg Goth, IEEE Software, Jan/Febr 2007

[BABEL] *PyPy - the ’Babelfish’ of programming*, Information Society Technologies Results, http://istresults.cordis.lu/index.cfm/section/news/tpl/article/ID/88611/BrowsingType/Features

[LEARN] *Sprint-Driven Development: working, learning and the process of enculturation in the PyPy community*, Avram, Sigfridsson, Sheehan, Sullivan, The Third International Conference on Open Source Systems, Limerick, Ireland, 2007

[FLOSS] *Successful OSS Project Design and Implementation: Requirements, Tools, Social Designs, Reward Structures and Coordination Methods* (To be published in Winter 2007, the submitted, not yet accepted chapter is titled "E(U)volving FLOSS communities - How FLOSS projects can stop worrying and learn to love public funding).

[REDDIT1] *blog post by a Ruby developer on the milestone release*, February 2007, http://programming.reddit.com/info/152lr/comments/c1560n

[D01.2-4] *Overview on Project Organization Activities*, PyPy EU-Report, 2007

[D02.1] *Configuration and Maintenance of Development Tools and Website*, PyPy EU-Report, 2007

[D02.2] *Introduce and Document a Release Scheme for PyPy Versions*, PyPy EU-Report, 2007

[D02.3] *Automated Testing and Development support library*, PyPy EU-Report, 2007

[D03.1] *Synchronization with Standard Python and the Extension Compiler*, PyPy EU-Report, 2007

[D06.1] *Core Object Optimization Results*, PyPy EU-Report, 2007

[D07.1] *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2007

[D08.1] *Release a JIT Compiler for PyPy Including Processor Backends for Intel and PowerPC*, PyPy EU-Report, 2007

[D08.2] *JIT Compiler Architecture*, PyPy EU-Report, 2007

[D09.1] *Constraint Satisfaction and Inference Engine, Logic Programming in Python*, PyPy EU-Report, 2007

[D10.1] *Aspect-Oriented, Design-by-Contract Programming and RPython static checking*, PyPy EU-Report, 2007

[D11.1] *A Case Study On Using PyPy for Embedded Devices*, PyPy EU-Report, 2007

[D12.1] *High-Level Backends and Interpreter Feature Prototypes*, PyPy EU-Report, 2007

[D13.1] *Integration and Configuration*, PyPy EU-Report, 2007

[D14.2] *Tutorials and Guide Through the PyPy Source Code*, PyPy EU-Report, 2007

[D14.3] *Report About Milestone/Phase 2*, PyPy EU-Report, 2007

[D14.4] *Report About Milestone/Phase 3*, PyPy EU-Report, 2007

[D14.5] *Documentation of the Development Process and Sprint Driven Development*, PyPy EU-Report, 2007

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

[ARCH] *PyPy architecture overview*, http://codespeak.net/pypy/dist/pypy/doc/architecture.html

[PYPYDOC] *PyPy online documentation*, http://codespeak.net/pypy/dist/pypy/doc/index.html