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

Final Activity Report

Authors and Contributors: Holger Krekel, Lene Wagner, Jacob Hallén, Beatrice During, Carl Friedrich Bolz, Laura Creighton, Armin Rigo, Michael Hudson, Samuele Pedroni, Christian Tismer, Alexandre Fayolle, Maciej Fijalkowski

Period covered: from 1 December 2004 to 31 March 2007

Date of preparation: 11th May 2007

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

Project Coordinator: Alastair Burt, Stephan Busemann

DFKI Saarbrücken **Revision: final**

Contents

1 PyPy: a Novel Open Source Dynamic Language Implementation Platform **3**

1.1 Introduction and Result Overview 3
1.2 PyPy - Vision and Relation to Existing Technology 3
1.3 PyPy - The Translation Framework 4
1.4 PyPy - the flexible Python Runtime Environment 5
1.5 PyPy - Sprint-Driven Development 6
1.6 Project Execution and Findings 8
1.7 Recommendations for EU/IST funded Open Source research 9
1.8 Conclusion and Outlook 10

2 Publishable Results **11**

2.1 The PyPy computer language implementation platform 11
2.2 The flexible Python interpreter 11
2.3 The py.test testing framework 11

3 Project Contact and Partner information **12**

1 PyPy: a Novel Open Source Dynamic Language Implementation Platform

1.1 Introduction and Result Overview

After 28 months of intense research, development and management activity, the final PyPy 1.0 milestone delivered all expected results and fulfilled its objectives.

PyPy breaks the compromise between flexibility, simplicity and speed for implementing today's dynamic computer languages. PyPy has challenged the common assumptions that the speed of a computer language is inversely proportional to its flexibility, and that simplicity must be sacrificed in order to obtain improvements in either. Using the PyPy platform, language interpreters can now be written at a higher level of abstraction, providing for flexibility and ease of implementation. PyPy's translation framework independently adds lower level aspects and produces versions of the interpreter for hardware or virtual target environments. Speed is recovered through various optimizations; a novel *Just in Time Compiler Generator* incorporates dynamic optimizations *automatically*. The alternative is hand-crafting complex and expensive code which is sensitively dependant upon many details of the interpreter, implementation and target environment.

PyPy's *flexible Python Interpreter* is a new self-hosted implementation of a widely used, dynamic, Very High Level Language. It is a complete and rather simple implementation, which has been tested against and made compliant with the 16 year old reference Python implementation, written in C. PyPy introduces state-of-the-art techniques in massive parallelization, security, distribution and persistence aspects in a compact and modular way and enables logic and aspect oriented programming extensions.

PyPy community members also found new uses for the project's tools and approaches, not foreseen in the original proposal. We went beyond the development of new interpreters to also produce new web servers, web applications and other special purpose programs which are efficiently and flexibly translated to various target environments including web browsers and small devices. The project maintains a web site with online examples of such applications (http://play1.pypy.org). The testing and debugging tools developed for the PyPy project have proven to be so useful that they have been adopted by many other projects and individuals worldwide.

PyPy's Sprint-Driven Development approach pioneered a new hybrid model for combining the contractual obligations of the EU Sixth Framework research funding with the Open Source culture, and the creation of a PyPy community that can sustain itself beyond the funding period. Nineteen week-long intense physical meetings ("sprints") in various international locations were the key factor for successful project development and for integrating and mentoring new contributors. PyPy's hybrid mix of distributed (or perhaps *dispersed*), agile and plan-driven development practices has become a research topic of its own and may serve as a blueprint for integrating EU funded research with Open Source engineering.

1.2 PyPy - Vision and Relation to Existing Technology

Traditionally, language interpreters are written in a target platform language such as C/POSIX, Java or C#. Each such implementation fundamentally 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 the Java virtual machine, is to provide standardized and higher level functionality in order to support language implementers in writing language implementations.

PyPy took a more ambitious approach. We defined a subset of the high-level language Python, called RPython, in which we implement languages as simple interpreters with few references to and dependencies on lower level details. Our translation framework then produces a concrete virtual machine for the platform of our choice by inserting appropriate lower level aspects. The result can be customized by selecting other feature and platform configurations.

Our vision is to provide a possible solution to the problem of language implementers: having to write $l \times o \times p$ interpreters for l dynamic languages and p platforms with o crucial design decisions. PyPy aims at having any one of these parameters changeable independently from each other:

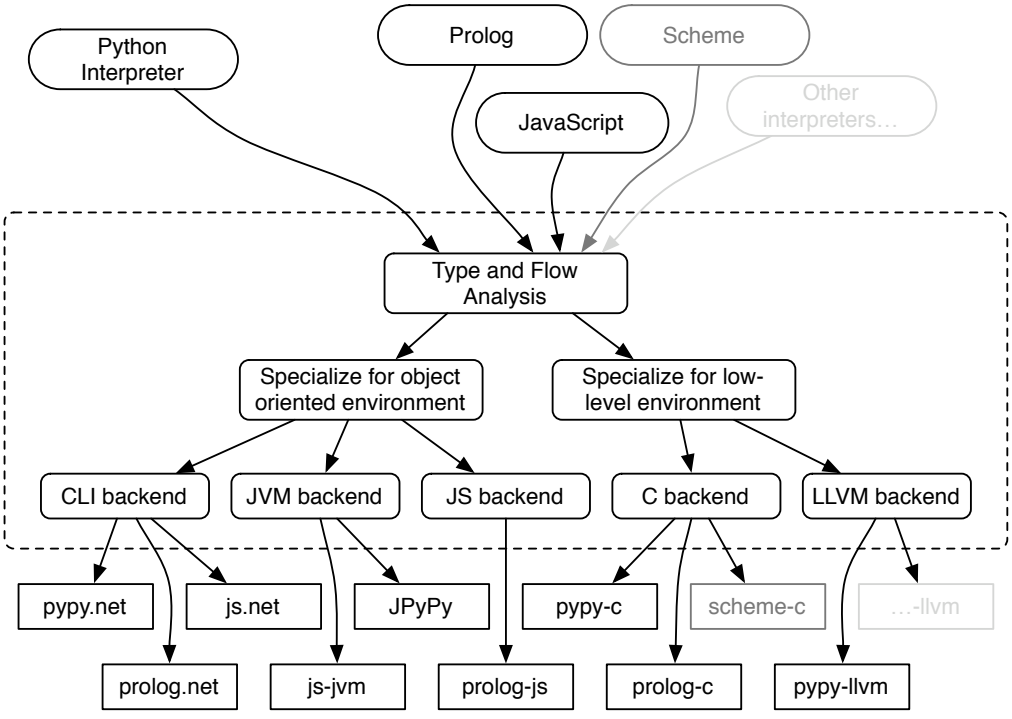
- l: the interpreter for the dynamic language can be evolved or entirely replaced;
- o: we can tweak and optimize the translation process to produce platform specific code based on different models and trade-offs;
- p: we can write new translator back-ends to target different physical and virtual platforms.

In contrast, a standardized target environment – say .NET – enforces $p = 1$ 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 research and engineering results give weight to this claim - at least as far as language implementation is concerned - showing an approach to the $l \times o \times p$ problem that does not rely on one-size-fits-all standardization.

1.3 PyPy - The Translation Framework

Traditional interpreter implementations make low-level choices early in the design process such as which memory management system to use, which environment to target and how to approach threading. Therefore these choices become often ingrained in the interpreter codebase and are very hard to change later in the lifetime of an implementation.

PyPy pioneers an approach where low level implementation choices are not made in the interpreter implementation but can be added through the metaprogramming techniques of the translation framework. Languages can thus be implemented at a conceptually higher level. New code generators for the translation framework allow the interpreter implementations to run in new target environments, even if not originally targeted by the implementer.



The low-level aspects that the interpreter implementation does not contain are inserted during the translation process. A good example for this is memory management. Since RPython is a garbage-collected language, the

language implementer does not have to deal with the error-prone process of managing memory manually. When an environment without automatic memory management is targeted (such as C), the translation process modifies the code to insert a garbage collector during translation. Several garbage collection approaches are available; such choices are configurable through a pervasive and systematic configuration system.

The idea of isolating memory management and garbage collection was extended to less obvious program transformations such as the so-called "Stackless transform". *Stackless Python* is a branch of the mainstream Python implementation that has existed for years, but has never been integrated into the main line of Python development. The major design problem in Stackless is to implement context switching in C, which needs some provision to leave the context of a function during its execution and to resume a different function where it was suspended. This is extremely hard to implement without manually rewriting many parts of the interpreter – if it is implemented at a low level such as C.

With PyPy, we solved this problem in its entirety as a translation aspect, supported by the Stackless transform, which automatically adds all the necessary house-keeping code for context switching. The Stackless implementation of PyPy is very small, reduced to the bare minimum of support code, and this has even created a new perception of the Stackless principles.

Being able to write an interpreter in a high-level language and to statically translate it into rather 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.

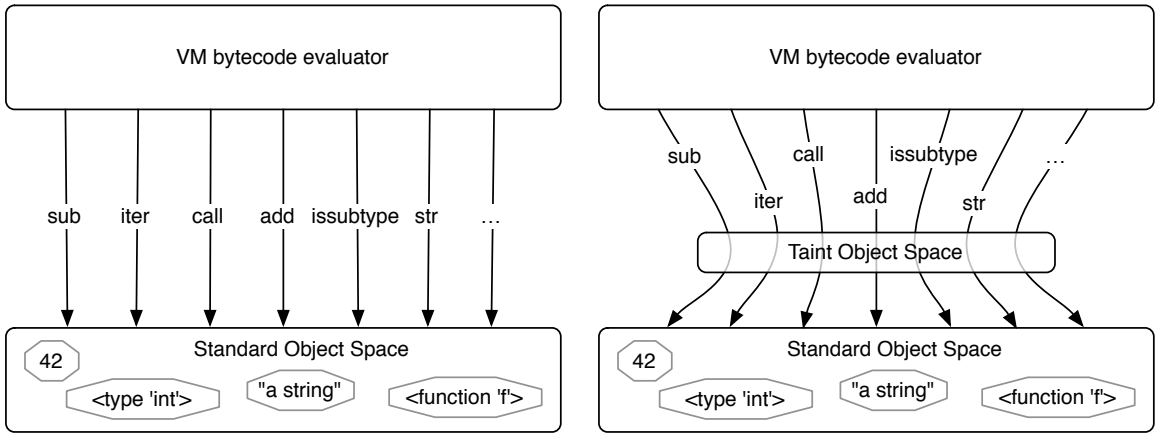
PyPy 1.0 includes the final piece of our vision: a novel practical approach to *generate Just-In-Time Compilers*, an area of language development that traditionally interweaves lower level implementation and language specification details. PyPy, by contrast, can create JIT compilers as another step during translation: 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 its final EU project milestone PyPy provides state-of-the-art JIT-typical speed benefits for algorithmic examples and represents the major research result of the project, breaking the previous compromise between flexibility, simplicity and speed for practical dynamic language implementations.

1.4 PyPy - the flexible Python Runtime Environment

The Python implementation is written in a high level language (RPython) as a relatively simple interpreter, easier to understand than CPython, the C reference implementation of Python. We successfully used RPython's high level of abstraction and flexibility to quickly experiment with features or implementation techniques in ways that in a traditional approach would require pervasive changes to the source code.

An important design decision was to separate the bytecode evaluation from the implementation of the various objects that the Python language offers. The interface of this object library (which is called an "object space") is the only way for the interpreter to access and manipulate Python application objects. The most important object space implementation is the "standard object space" which implements the object semantics exactly as the Python language demands.

This separation between the bytecode evaluation and the object semantics is very important because it allows one to generically add and amend the behaviour of application objects. We have thus created new object spaces, usually by extending the standard object space in a systematic manner through meta-programming techniques. A good example of this is our "Taint" object space, a successful experiment which adds information flow security features to prevent the leakage of sensitive information. The "Taint" object space proxies all operations on application objects and can easily tag and trace information flow within an application. At a review workshop at IBM in Zürich, scientists acknowledged that we have already accomplished the state-of-the-art with our prototype.



We also worked on adding language level support which makes remote objects appear as local ones, provides orthogonal persistence, experiments with lazy evaluation features and adds logical variables to Python. In all cases, no pervasive changes to the code base were required. All prototype features could be implemented with a few hundred lines of code, through a quick test-driven development cycle which systematically enhanced the standard object space.

Support for aspect-oriented programming (AOP) was added through a high level API which can change language syntax at runtime. Logic programming was introduced to Python by providing logic variables and by porting a constraint solver package to RPython. With this an OWL (the Web Ontology Language) could be implemented that converts the OWL ontology into a constraint problem, and queries can be made using the SPARQL query language.

All of the implementation aspects and extensions are expressed in a modular way and with a high level of abstraction. PyPy's translation framework produces interpreter versions for many target environments: currently C, .NET and the Low Level Virtual Machine Compiler Infrastructure (LLVM). This makes PyPy highly suitable for the rapid development of new language features and extensions and new models of middleware application development - as well as for teaching modern concepts of language development.

1.5 PyPy - Sprint-Driven Development

PyPy, as an Open Source project, does not only consist of the EU consortium but also of a growing community of interested developers, scientists and contributors around it. Since its start (and before it recieved EU-funding), PyPy has been carried forward by this community, with its own interests and contributions. The EU project, has been dedicated to leveraging and growing this potential in addition to satisfying its technical and scientific objectives, as well as reacting to needs and incorporating ideas expressed by the community. A further cornerstone of the PyPy development is that the core team itself has worked distributed all over Europe, while collaborating closely on almost every area of the project on a day to day basis.

During the entire project, *sprints* - week long co-located coding sessions every 6-8th week, were the key driving factor of the project. In total, 19 sprints were arranged across Europe, in the US and Japan. They served to integrate and advance technical work as well as to integrate and mentor newcomers. In the phases between the sprints, the team worked distributed, coordinating via electronic channels, mostly using mailing lists and IRC (Internet Relay Chat). The project also maintained and published extensive online documentation, reports, tutorials and video documentation.

In order to succeed in implementing Sprint-Driven Development, as well as to facilitate dissemination and new contributors, an efficient software development infrastructure has been a critical requirement. PyPy is built on thousands[1] of *automated tests* in all its coding areas, being run regularly to ensure the quality and consistency of the codebase. This test driven approach ensures that remote changes do not invisibly disrupt other functionality. This lowers the entry barrier for newcomers, who can experiment with changes without challenging the project's integrity. The tests are driven by a custom tool, `py.test`, which has been separately released and is used by many other projects and organisations.

The project refined its sprinting approach during the EU project, pre-targeting the sprints either at reaching important internal milestones or at introducing newcomers. Sprints also increasingly served to bring 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 campaign, in which PyPy core developers participated as mentors in 2005 and 2006, PyPy launched its Summer of PyPy 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 ideas and then mentoring and supporting their efforts was an attractive incentive and enlarged the group of core developers. The work on the proposals initiated the implementation of a JavaScript interpreter, a JavaScript/AJAX tool for generating web applications, a .NET/CLI backend, a JVM backend and several other contributions. Newcomers not only entered through PyPy sprints, but also through more traditional means of Open Source development: IRC channels and mailing lists.

The unique development practice of Sprint-Driven Development has been monitored closely, presented to academic and industrial audiences and introduced into the context of Agile development research. PyPy published experience reports at the two main Agile Community conferences in Europe and the US. The resulting validation from the Agile community resulted in feedback stating that Sprint-Driven Development could be seen as a recommended best practice in order to manage Agile XP-projects that are facing a distributed project context - an area where the Agile community is struggling today.

Interest in the practice and the community effects of its implementation within PyPy also spread to other EU-projects, foremost CALIBRE. The PyPy/CALIBRE collaboration resulted in international conference participa-

[1] as we write this, the number of automated tests is 11805.

tion, sprint hosting and further research, participation and arranging of workshops. Concrete examples of this collaboration would be the seminar for the Commission in December 2006: "Best Practice in the Use and Development of Free and Open Source Software" and the invitation to participate in the internal CALIBRATION workshop at Philips Medical System. Parts of the research community (contacts established through the CALIBRE collaboration), studying the F/OSS phenomenon took a closer look at the sprint practice as implemented in PyPy and published their findings.[2]

1.6 Project Execution and Findings

1.6.1 Contractual Research and Open Source Development

Traditional academic Computer Science research resolves around the publication of papers and is often measured by the number of papers published, by how prestigious one's place of publication and how often one is cited by other members of the community.

Open Source developers publish, and publish often, but what they publish is code or prototypes. Success is measured by how many people are interested and how many are using the code, though these numbers can never be known with any great accuracy. There is a strong perspective of providing something that is both useful and interesting.

PyPy as well as other research oriented Open Source projects share the goal of providing useful practical research.

During the EU funding period, PyPy had to integrate a quite diverse set of cultural and formal factors - on the one hand an Open Source community based on volunteering, active and demanding contributors, on the other hand an IST research funding contract with contractual obligations, scientific goals, deliverables and timelines.

In practice, this meant we had to integrate mentoring and teaching efforts with technically complex and time-critical contractual tasks. From the current perspective, PyPy could only become a successful project because it managed to adapt and form a fruitful compromise, its methodology, its management, its dissemination and research focuses taking its specific context into account. PyPy as a project has been radically different both from a 'normal' EU project as well as a 'traditional' Open Source one.

The technical management and the consortium management has adapted to the dynamic project implementation by behaving in a more and more 'agile' way over time. During some periods, the technical implementation evolved in an atmosphere of informal collaboration where the contractual obligations of the work packages were the only constraints. The resource consumption and the progress of development has been monitored by the management at frequent intervals. Parts of the management have been involved in the technical implementation, too, which helped the management to keep close track of the actual ongoings on the development level.

Work packages and time plans have been modified to incorporate evolving ideas and handle new requirements without compromising the contractual objectives. Fortunately, the Commission reacted with flexibility when we requested changes to the work plan, project duration and also prepayment calculations, thus supporting and enabling a smooth implementation.

1.6.2 Open Source Community involvement, Summer of PyPy

For encouraging and involving the Community, we considered it important to give as many single contributors as possible the opportunity to be involved in the EU contract implementation, including being reimbursed for some of their expenses. However, although the Commission supported this intention, it was very hard to find a model that worked with the financial guidelines of FP6. Four qualified contributors acceded the EU contract as "Physical Partners", requiring contract amendments and the person to have a full share in the consortium duties

[2] The socGDS team at University of Limerick, "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.

towards the EU – far too heavy an obligation for experts that only contribute one or a few intense work weeks and only receive travel reimbursement.

The project discussed a new model with the Commission at the interim review and initiated the Summer of PyPy campaign. A university partner set up the program for student participation, a lightweight procedure to reimburse travel expenses to sprints was found. Once in place, this was very successful.

Positive side-effect of the Summer-of-PyPy campaign has been that we directly targeted undergraduate and post-graduate students when disseminating and involving new contributors. We consider that important, as they are the carriers of future research and future community results as well as being very suitable for further dissemination within the university groups themselves.

1.6.3 Small Enterprises as driving forces, transnational contracting

Unlike most EU projects, the PyPy consortium has had a majority of SME organisations, who have employed experts from the community to implement the project. The co-financing approach of the commission, only reimbursing 50% of costs was burdensome for these companies. It can be hard for SME's to focus on the long-term goals of serious research and development work, while simultaneously having to make money to finance its own share of the costs.

Therefore, we highly endorse the implementation of a higher funding rate for SME's with The Seventh Framework Programme.

The project partners have always had a strong interest in attracting the most talented people to do PyPy research, without regard for where in Europe they live. But the Sixth Framework still assumes that most people work in the country where they reside. The growth of transnationals, out-sourcing, and internet consulting has made this assumption less and less true over time. Because of huge differences in taxation, labour laws and general lack of support for the case where a resident of one EU nation actually works for an employer in an other country, we have on several occasions had to forgo hiring people that would have improved project performance.

In the computer field, it is now standard practice for companies to hire such people as consultants rather than employees. The PyPy project would have liked to do the same, but by the definitions established by the Sixth Framework Programme, this common practice is considered sub-contracting and has a number of restrictions; it is hard to use this contracting model in practice.

1.7 Recommendations for EU/IST funded Open Source research

In summary, we think that the successful conclusion of the PyPy project shows that it is not only possible to combine EU/IST funding and Open Source development, but that when done with care it is greatly to the interest of both communities. Furthermore, we think that some of our findings may be helpful for successful future cooperation and would therefore like to present some recommendations based on our findings.

Our project experience suggests that Open Source projects considering EU/IST research funding would do well to:

- consider an agile sprint-driven model of development;
- take care to share some funding with upcoming contributors;
- consciously balance contractual with community interests and consider changes to the contractual work plan if it fits the objectives better;
- implement a transparent and flat communication and decision culture, be ready to adjust roles and not stick to formal issues too much.

We recommend that EU/IST officials considering funding research oriented Open Source projects:

- support models to allow Open Source projects to flatly share funding with upcoming contributors in a lightweight way;
- encourage modifications to contractual work plans if it serves Open Source community interests;
- consider how to ease transnational hiring of experts;
- consider how Open Source development and scientific research increases the applicability, availability and impact of results.

1.8 Conclusion and Outlook

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 Sun's JVM or Microsoft's .NET environments. During the last project phase, the PyPy team took care to open up to non-Python developers and received encouraging feedback. The evolving JavaScript interpreter, a new Scheme interpreter implementation and a refined Java backend, the latter two now funded through Google's Summer of Code program, mark PyPy's way to becoming a general platform for practically implementing dynamic languages.

PyPy's *Python interpreter* delivers many features not present in other Python implementations. Yet, it has rough edges that can be expected from a research effort and thus engineering efforts remain to gain more mainstream adoption. However, with PyPy's increasing visibility as a flexible and fast platform and its compliant and rich Python interpreter, it should not be hard to gather community focus around removing these remaining engineering obstacles.

PyPy's *Open Source community* consists of many individuals who work independently or have been contracted through a company participating in various PyPy development areas. After the expansive 28 month of European Union's funding phase PyPy's future will likely be structured in a more typical Open Source style: advances will be driven by somewhat separate interests, namely engineering, academic and commercial interests. However, the involved people and organisations share a common interest in moving PyPy forward and intend to collaborate on upcoming tasks or contracts. A group of trusted individual contributors has been installed to keep conceptual integrity on the technical level and in relation to upcoming funding opportunities.

PyPy's code base provides an open and very fertile ground for many more developments to come. Research interest mainly lies in PyPy as a platform for implementing dynamic languages in general and in introducing novel features through interpreter architecture improvements and experiments. More specifically, some academically interested people involved in the PyPy project plan to focus on Just in Time compiler generation as one of the major topics that invite in-depth exploration. Other groups have already independently started to work on other aspects related to PyPy's approach or implementation techniques.

Commercial interest will continue to focus on using PyPy's powerful translation tool chain and its high level and fast RPython language. Programming in this language allows complex programs to be quickly developed and tested, and then automatically get versions that fit well with machine-level or virtualized environments. Apart from the advanced and flexible Python runtime environment, writing special purpose interpreters or generating web applications are of commercial interest.

PyPy shows that a good combination of Open Source development and scientific research increases applicability, availability and impact of results. Despite the open license which allows anyone to use and exploit PyPy's results, the consortium partners and involved contributors have an advantage in that they can more easily build products and services on top of the environment that PyPy provides.

To keep a balance between individual, research, commercial and community interests, organising sprints will remain as a key approach to keeping conceptual integrity and bringing together PyPy developers, newcomers and interested parties.

2 Publishable Results

The PyPy EU project produced three major exploitable results: the PyPy computer language implementation platform, a flexible Python interpreter written using this platform, and the py.test testing framework.

2.1 The PyPy computer language implementation platform

Initially constructed as an implementation of the Python interpreter in Python, PyPy is now a platform supporting multiple different dynamic languages, with Python, JavaScript and Prolog currently implemented. Pervasive use of metaprograming and advanced methods for translation and just-in-time optimizations provide high performance on a number of backend systems. There is support both for low level backends like C/POSIX and LLVM (Low Level Virtual Machine) as well as high level ones, including CLI/.NET, Smalltalk and JVM.

The platform is still in a development stage, with the Python/C/POSIX combination approaching readiness for production use. All parts of the platform are published under the MIT Open Source License.

The PyPy community welcomes anyone wishing to contribute to the further development of the platform. Several of the project partners are willing to accept contracts for the further development of various aspects of the platform.

For software download, documentation and contact information, please see http://pypy.org.

2.2 The flexible Python interpreter

The initial and most mature interpreter written in the PyPy implementation platform is a fully compliant interpreter for the Python 2.4 language. It is more flexible and open to language research and enhancements than any pre-existing implementation of Python, supporting novel features that give programmers better solutions to existing problems (security, distribution, parallelism, logic programming, etc.).

The interpreter is mature and available under the MIT Open Source License.

For download, documentation and contact information, please see http://pypy.org. Precompiled binary versions are available from the web site or as Debian packages.

2.3 The py.test testing framework

An advanced and flexible testing tool written in the Python programming language. Compared to other unit testing frameworks for Python, it requires much less boiler plate code, gives better control over the testing process and makes use of idioms that are closer to the Python programming philosophy. It provides cross-platform and distributed testing extensions.

The framework is mature and is available under the MIT Open Source License.

merlinux GmbH provides commercial support and consultancy services for the deployment and further development of the framework. For software download, documentation and contact information, please see http://merlinux.de and http://codespeak.net/py.

3 Project Contact and Partner information

Project Contact Data	
Project Web site	http://pypy.org
Press Contact	Beatrice Düring <bea@changemaker.nu>, Holger Krekel <office@merlinux.de>
Consortium Co-ordinator	Stephan Busemann <busemann@dfki.de>
Consortium Project Management	pypy-manage@codespeak.net
Consortium	pypy-funding@codespeak.net
Developer/Contributor core group	pypy-ct@codespeak.net

Consortium Partners of the EU/IST 04779 project:

- DFKI GmbH, Saarbrücken, Germany
- Open End, Göteborg, Sweden
- Logilab, Paris, France
- Change Maker, Göteborg, Sweden
- merlinux GmbH, Hildesheim, Germany
- tismerysoft GmbH, Berlin, Germany
- Heinrich Heine Universität Düsseldorf, Düsseldorf, Germany
- Impara Gmbh, Magdeburg, Germany
- Physical Partners: Laura Creighton, Richard Emslie, Niklaus Haldimann, Eric van Riet Paap