PyPy 1.0 and beyond

PyCon Uno 07, Firenze (Italy)

Antonio Cuni cuni@disi.unige.it
Università degli Studi di Genova

based on Michael Hudson's talk at PyCon 2007

What you're in for in the next 45 mins

- Quick intro and motivation
- Quick overview of architecture and current status
- Introduction to features unique to PyPy, including the JIT
- A little talk about what the future holds

What is PyPy?

- PyPy is:
  - An implementation of Python, and a very flexible compiler framework (with some features that are especially useful for implementing interpreters)
  - An open source project (MIT license)
  - A STREP (“Specific Targeted REsearch Project”), partially funded by the EU
  - A lot of fun!

30 second status

- We can produce a binary that looks more and more like CPython to the user
- 2-4x slower, depending on details
- More modules supported – socket, mmap, …
- Can now produce binary for CLI (i.e. .NET)
- Can also produce more capable binaries – with JIT, stackless-style coroutines, with logic variables, transparent proxies, ...

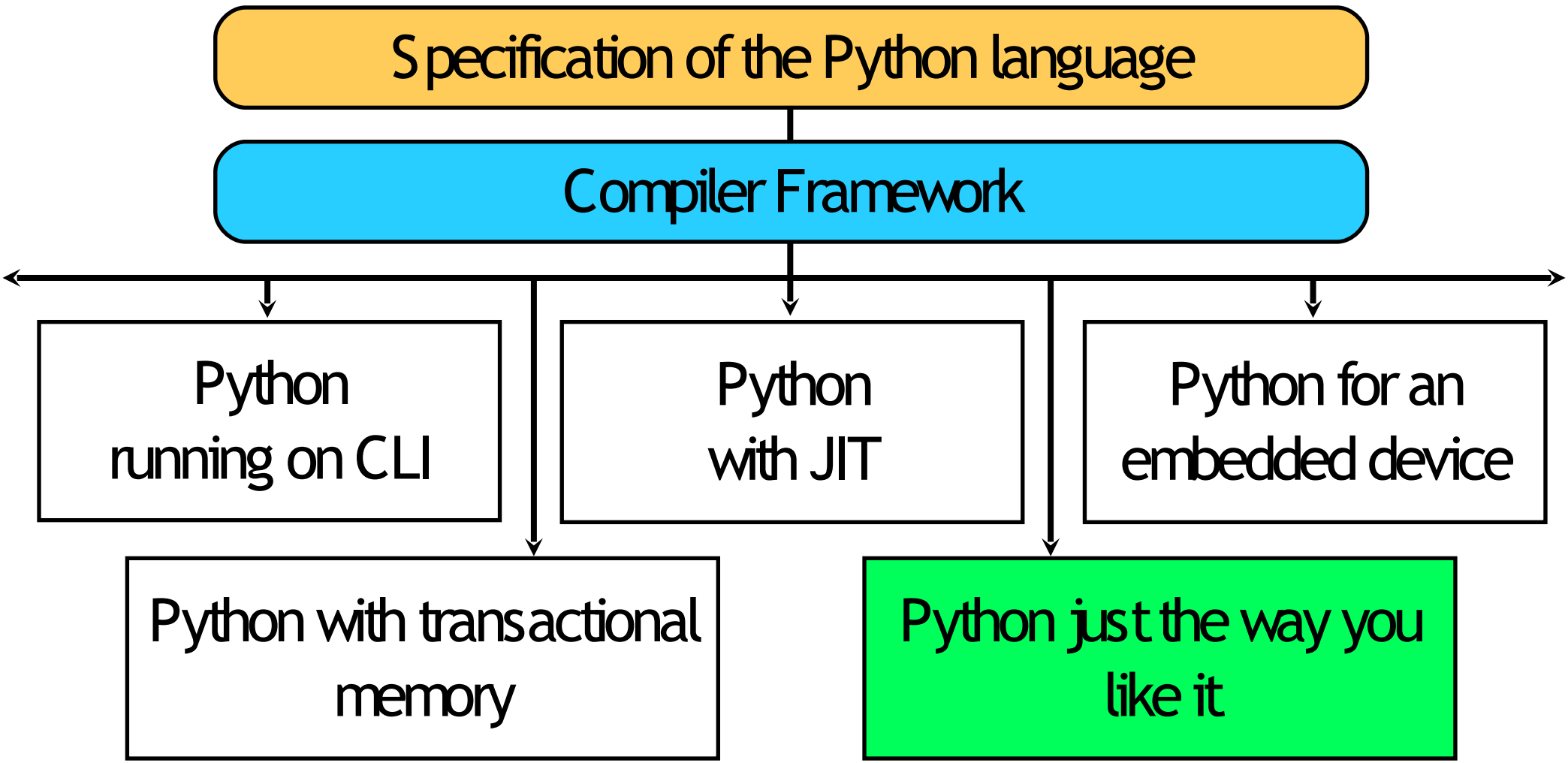
Motivation

- PyPy grew out of a desire to modify/extend the *implementation* of Python, for example to:
  - increase performance (psyco-style JIT compilation, better garbage collectors)
  - add expressiveness (stackless-style coroutines, logic programming)
  - ease porting (to new platforms like the JVM or CLI or to low memory situations)

Lofty goals, but first...

- CPython is a fine implementation of Python but:
  - it's written in C, which makes porting to, for example, the CLI hard
  - while psyco and stackless exist, they are very hard to maintain as Python evolves
  - some implementation decisions are very hard to change (e.g. refcounting)

Enter the PyPy platform



How do you specify the Python language?

- The way we did it was to write an interpreter for Python in **RPython** – a subset of Python that is amenable to analysis
- This allowed us to write unit tests for our specification/implementation that run on top of CPython
- Can also test entire specification/implementation in same way

py.py demo

```
antocuni@anto pypy $ ./bin/py.py
PyPy 1.0.0 in StdObjSpace on top of Python 2.4.4
>>>> import sys
>>>> print sys.version
2.4.1 (pypy 1.0.0 build 43973)
```

py.py demo

```
antocuni@anto pypy $ ./bin/py.py
PyPy 1.0.0 in StdObjSpace on top of Python 2.4.4
>>>> import sys
>>>> print sys.version
2.4.1 (pypy 1.0.0 build 43973)
>>>>
>>>> myint = 42
>>>> mystring = “Hello world”
>>>> <ctrl-C>
```

py.py demo

```
antocuni@anto pypy $ ./bin/py.py
PyPy 1.0.0 in StdObjSpace on top of Python 2.4.4
>>>> import sys
>>>> print sys.version
2.4.1 (pypy 1.0.0 build 43973)
>>>>
>>>> myint = 42
>>>> mystring = “Hello world”
>>>> <ctrl-C>
Python 2.4.4 (#1, Jan 13 2007, 15:05:52)
[GCC 4.1.1 (Gentoo 4.1.1)] on linux2
*** Entering interpreter-level console ***
>>> w_myint
W_IntObject(42)
>>> w_mystring
W_StringObject('hello world')
```

py.py -o thunk

```
$ py.py -o thunk
PyPy 1.0.0 in ThunkObjSpace on top of Python 2.4.4
>>>> def f():
....     print 'computing...'
....     return 6*7
....
>>>>
```

py.py -o thunk

```
$ py.py -o thunk
PyPy 1.0.0 in ThunkObjSpace on top of Python 2.4.4
>>>> def f():
....     print 'computing...'
....     return 6*7
....
>>>> from __pypy__ import thunk
>>>> x = thunk(f)
>>>> x
computing...
42
>>>> x
42
```

py.py -o thunk

```
$ py.py -o thunk
PyPy 1.0.0 in ThunkObjSpace on top of Python 2.4.4
>>>> def f():
....     print 'computing...'
....     return 6*7
....
>>>> from __pypy__ import thunk
>>>> x = thunk(f)
>>>> x
computing...
42
>>>> x
42
>>>> y = thunk(f)
>>>> type(y)
computing...
<type 'int'>
```

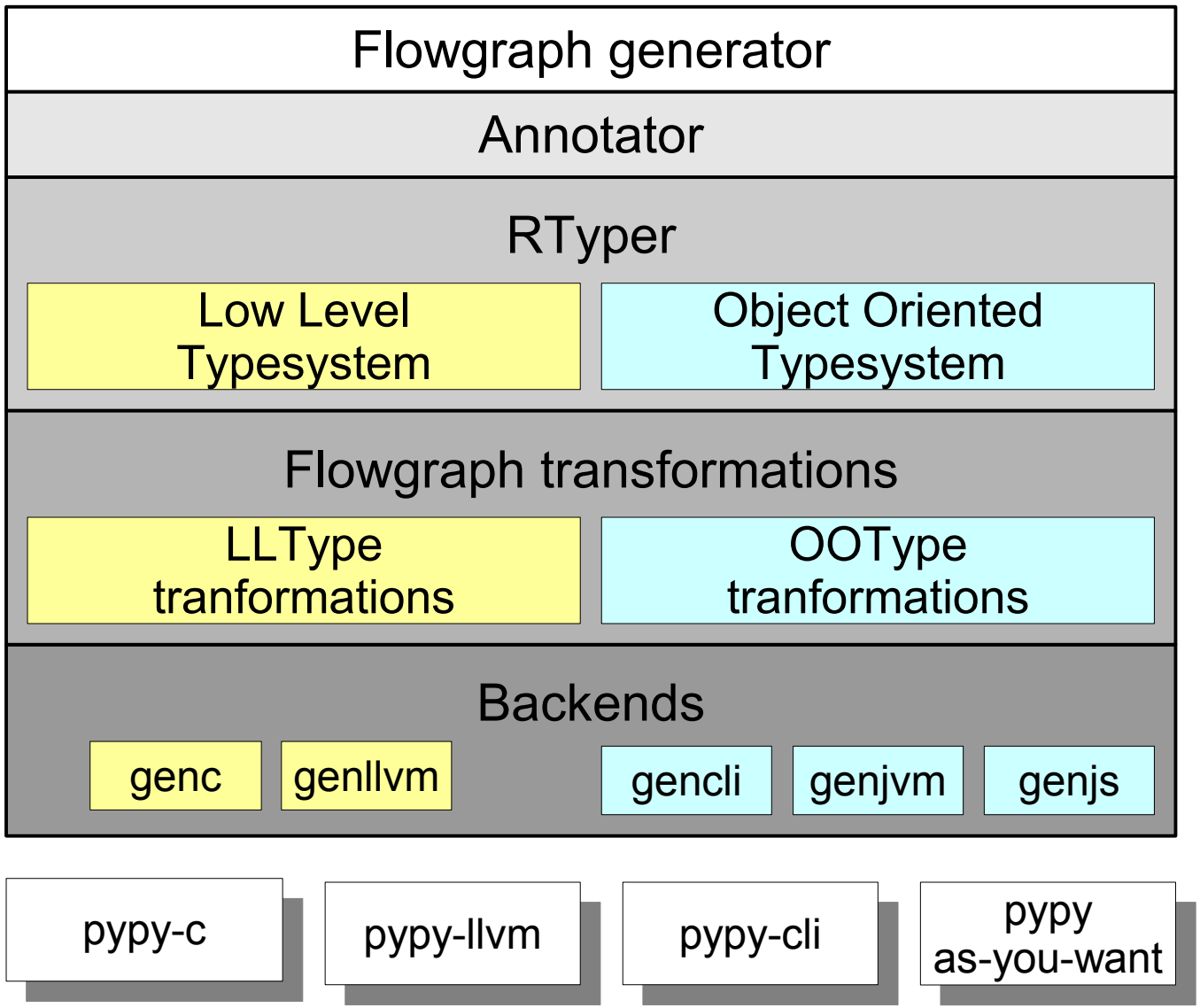
py.py -o thunk

```
$ py.py -o thunk
PyPy 1.0.0 in ThunkObjSpace on top of Python 2.4.4
>>>> from __pypy__ import lazy
>>>> @lazy
>>>> def fibo(a, b):
....     return (a, fibo(b, a + b))
....
>>>> fibonacci = fibo(1, 1)
>>>> import pprint
>>>> pprint.pprint(fibonacci, depth=8)
(1, (1, (2, (3, (5, (8, (13, (21, (34, (...)))))))))
```

Translation Aspects

- Our Python implementation/specification is very high level
- One of our Big Goals is to produce our customized Python implementations without compromising on this point
- We do this by weaving in so-called ‘translation aspects’ during the compilation process

RPython compiler



RPython compiler

Specification of the Python language

Flowgraph generator

Annotator

RTyper

Low Level Typesystem

Object Oriented Typesystem

Flowgraph transformations

LLType tranformations

OOType tranformations

Backends

genc

genllvm

gencli

genjvm

genjs

pypy-c

pypy-llvm

pypy-cli

pypy as-you-want

RPython compiler

Specification of the Python language

Flowgraph generator

Annotator

RTyper

Low Level Typesystem

Object Oriented Typesystem

Flowgraph transformations

LLType tranformations

OOType tranformations

Backends

genc

genllvm

gencli

genjvm

genjs

Javascript

Prolog

Scheme

????

pypy-c

pypy-llvm

pypy-cli

pypy as-you-want

The Annotator

- Works on control flow graphs of the source program
- Type annotation associates variables with information about which values they can take at run time
- An unusual feature of PyPy's approach is that the annotator works on live objects which means it never sees initialization code, so that can use `exec` and other dynamic tricks

The Annotator

- Annotation starts at a given entry point and discovers as it proceeds which functions may be called by the input program
- Does not modify the graphs; end result is essentially a big dictionary
- Read “Compiling dynamic language implementations” on the web site for more than is on these slides

The RTyper

- The RTyper takes as input an annotated RPython program (e.g. our Python implementation)
- It reduces the abstraction level of the graphs towards that of the target platform
- This is where the magic of PyPy really starts to get going :-)

The RTyper

- Can target a C-ish, pointer-using language or an object-oriented language like Java or Smalltalk with classes and instances
- Resulting graphs are not completely low-level: still assume automatic memory management for example

Reducing Abstraction

- Many high level operations apply to different types – the most extreme example probably being calling an object
- For example, calling a function is RTyped to a "`direct_call`" operation
- But calling a class becomes a sequence of operations including allocating memory for the instance and calling any `__init__` function

Further Transforms

- RTyping is followed by a sequence of further transforms, depending on target platform and options supplied:
  - GC transformer – inserts explicit memory management operations
  - Stackless transform – inserts bookkeeping and extra operations to allow use of coroutines, tasklets etc
  - Various optimizations – malloc removal, inlining, ...

The Backend(s)

- Maintained backends: C, LLVM, CLI/.NET, JVM and JavaScript
- All proceed in two phases:
  - Traverse the forest of rtyped graphs, computing names for everything
  - Spit out the code

Status

- The Standard Interpreter almost completely compatible with Python 2.4.4
- The compiler framework:
  - Produces standalone binaries
  - C, LLVM and CLI backends well supported, JVM very nearly complete
  - JavaScript backend works, but not for all of PyPy (not really intended to, either)

Status

- The C backend support “stackless” features – coroutines, tasklets, recursion only limited by RAM
- Can use OS threads with a simple “GIL-thread” model
- Our Python specification/implementation has remained free of all these implementation decisions!

What we're working on now

- The JIT
  - i386, PowerPC and LLVM backends
- Object optimizations
  - Dict variants, method caching, …
- Integration with .NET
- Security and distribution prototypes
  - Not trying to revive rexec for now though…

Things that make PyPy unique

- The Just-In-Time compiler (and the way it has been made)
- Transparent Proxies
- Runtime modifiable Grammar
- Thunk object space
- JavaScript (demos: b-n-b and rxconsole)
- Logic programming

RPython as a language

- Designed as an implementation detail
- Could be useful of his own
- Less convenient than Python...
- ...but much faster (up to 300% faster than CPython)
- Create extension module for CPython (extcompiler)
- Create .NET exe/dll (in-progress)

JIT demo

```
def f1(n):
    i, x = 0, 1
    while i<n:
        j = 0
        while j<=i:
            j = j + 1
            x = x + (i&j)
        i = i + 1
    return x

try:
    import pypyjit
except ImportError:
    print "No jit"
else:
    pypyjit.enable(f1.func_code)
```

JIT demo

```
$ python demo/jit/f1.py
No jit
1083876708
5 iterations, time per iteration: 0.695304632187
```

JIT demo

```
$ python demo/jit/f1.py
No jit
1083876708
5 iterations, time per iteration: 0.695304632187

$ ./pypy-c demo/jit/f1.py
1083876708
5 iterations, time per iteration: 0.0104030132294
```

JIT demo

```
$ python demo/jit/f1.py
No jit
1083876708
5 iterations, time per iteration: 0.695304632187

$ ./pypy-c demo/jit/f1.py
1083876708
5 iterations, time per iteration: 0.0104030132294

$ python -c 'print 0.6953 / 0.0104'
66.8557692308
```

About the project

- Open source, of course (MIT license)
- Distributed – developers are all around the world (mostly in Europe)
- Sprint driven development – focussed week long coding sessions. Every ~6 weeks during funding period, less frequently now.
- Extreme Programming practices: pair programming, test-driven development

Future Facts

- Funding period ended March 31st
- So PyPy development ended? Of course not!
  - PyPy was a hobbyist open source project before funding, will return to that state
  - … for a while, at least

Future Hopes

- At least in my opinion, the work so far on PyPy has mostly been preparatory – the real fun is yet to come.
- Likely future work includes:
  - More work on the JIT
  - Reducing code duplication
  - Improved C gluing, better GIL handling
  - Better interpreter for CLI and JVM

Future Dreams

- High performance compacting, generational, etc GC (steal ideas from Jikes?)
- Implementations of other dynamic languages such as JavaScript, Prolog (already started), Scheme (Google SoC), Ruby (?), Perl (??) (which will get a JIT essentially for free)
- The ability to have dynamically loaded extension modules

Join the fun!

- Project relies more than ever on getting the community involved
- Read documentation:

  http://codespeak.net/pypy/

- Come hang out in #pypy on freenode, post to pypy-dev
- Probably will be easier to keep up now…
- EuroPython post-sprint!

Thanks for listening!

Any Questions?