An Introduction to PyPy

KiwiPyCon 2009

Michael Hudson micahel@gmail.com
Canonical Ltd.

What you're in for in the next 25 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 in Python
  - A very flexible compiler framework (with some features that are especially useful for implementing interpreters)
  - An open source project (MIT license)
- PyPy was partly EU-funded from 2005-2007

What we've got

- We can produce a binary that looks very much like CPython to the user
- Some, but not all, extension modules supported – socket, mmap, termios, …
- Can produce binary for CLR/.NET (watch out IronPython! :-) and JVM (ditto Jython…)
- Can also produce binaries with more features (stackless, JIT, …)

Motivation

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

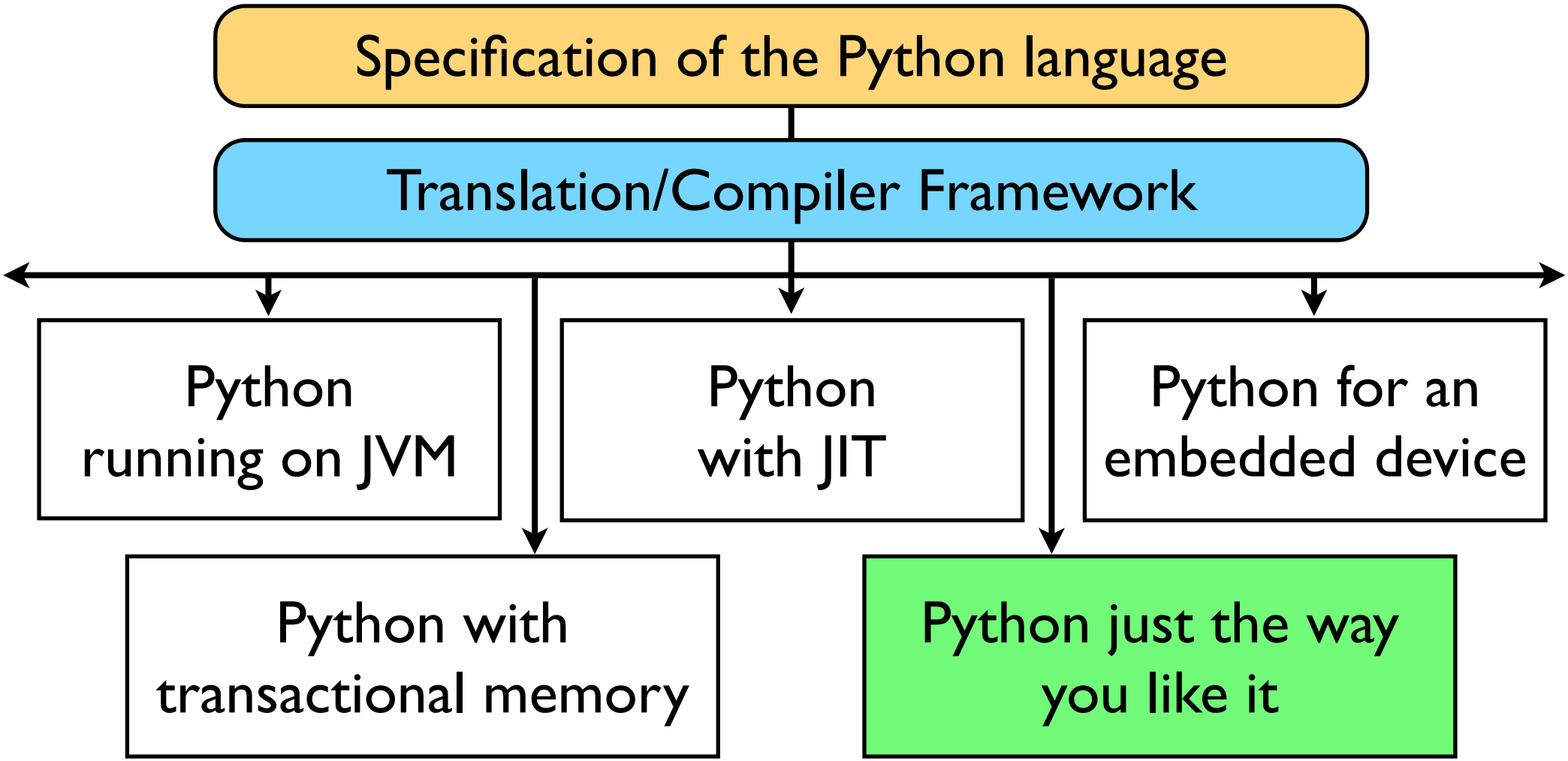
Problems with CPython

- 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)

PyPy's Big Idea

- Take a *description* of the Python programming language
- Analyze this description:
  - Decide whether to include stackless-like features or a JIT
  - Decide which GC to use
  - Decide the target platform
- Translate to a lower-level, efficient form

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

The Translation/ Compiler Framework

- The compiler framework takes as input live Python objects (as opposed to source code)
- It *abstractly interprets* the bytecode of functions to produce flow graphs
- Further layers of abstract interpretation perform more analysis and gradually reduce the level of abstraction
- Finally C or other source code is generated

If you have a hammer…

- We'd written this compiler framework, with only one expected non-trivial input (our Python interpreter)
- We realized that it would be suitable for implementations of other dynamically-typed programming languages
- Now have implementations of Prolog, Smalltalk, JavaScript and Scheme (to varying extents)

The *L*×*O*×*P* problem

This leads to one of PyPy's meta-goals, ameliorating the so-called *L*×*O*×*P* problem: given

- *L* dynamic languages
- *O* target platforms
- *P* implementation decisions

we don't want to have to write *L*×*O*×*P* different interpreters by hand.

The *L*×*O*×*P* problem

- PyPy aims to reduce this to an *L*+*O*+*P* problem:
  - Implement *L* language front-ends
  - Write backends for *O* platforms
  - Take *P* implementation decisions
- Then let the *magic of PyPy*™ tie it all together :-)

The exciting bit – the JIT

- Most recent work has been on the second (depending how you count) version of the JIT
  - First incarnation sort of worked, but was completely crazy
  - It transformed the control flow graph of an interpreter into that of a compiler
- This attempt based on the idea of a "tracing JIT" like TraceMonkey or Tamarin

The exciting bit – the JIT

- It finds "traces" – frequently executed sequences of bytecode – in the program
- Once a trace has been identified, it is interpreted in "tracing mode" where a record is kept of the low level operations that would be executed interpreting the bytecode
- This is then compiled to machine code and executed

Status – Interpreter

- PyPy's Python interpreter supports 2.5.2 by default
- 2.6 should be easy enough
- No Py3K yet :-) (will be work, but not too insanely hard)
- The "__pypy__" module includes a variety of mysterious and exciting things, depending on options supplied

Status – compiled interpreter

- When compiled to (standalone) C with all the optimizations turned on, our interpreter varies from a little faster to about twice as slow than CPython
- Can also build interpreters with threading and with stackless features

Status – backends

- We currently have three complete backends:
  - C/POSIX (like CPython)
  - CLI (like IronPython)
  - JVM (like Jython)

Status – the JIT

- Usefulness of the JIT varies from program to program
  - Good examples are at least 50 times faster than CPython
  - Bad ones still twice as slow
- Supports ia32, amd64 on the way
- Can also generate CLI bytecodes (which will then be JITted by Mono or the CLR)

Extra: sandboxing

- Something completely different!
- You can build PyPy's Python interpreter in a way that can use limited CPU, memory and system calls
  - Works by running modified interpreter as a subprocess of a trusted "monitor"
  - All system calls in the interpreter are replaced with code to request the monitor perform the call

Future Work?

- JIT JIT JIT: making it make more practical:
  - speed up more programs by more
  - cap memory usage
- Easier platform integration
  - Particularly for JVM and CLI backends
- Implementations of other dynamic languages (which will get a JIT essentially for free)?
- Experiment with removing the GIL??

About the Project

- Open source, of course (MIT license)
- Read documentation:

  http://codespeak.net/pypy/

- Project has a somewhat academic focus compared to most open source – lots of papers to read!
- Come hang out in `#pypy` on freenode, post to `pypy-dev@codespeak.net`

Thanks for listening!

Any Questions?