PyPy status talk

Maciej Fijalkowski
Merlinux GmbH

Politechnika Wroclawska

January 22 2009

What this talk is about?

- general overview of dynamic languages vm
- example: python
- challenges of classic approach
- possible solution - pypy

Dynamic languages VMs

- written in lower level language (C, Java)
- usually hard coded design decisions (eg about GC, object layout, threading model)
- hard to maintain
- a challenge between performance and maintainability

Example - python

- primary implementation - CPython
- written in C
- hard-coded - Global Interpreter Lock
- hard-coded - refcounting for garbage collection
- psyco - very hard to maintain

Example - python (2)

- Jython, IronPython - bound to a specific VM
- about the same performance as CPython
- Java is still not the best language ever
- both are compilers, harder to maintain

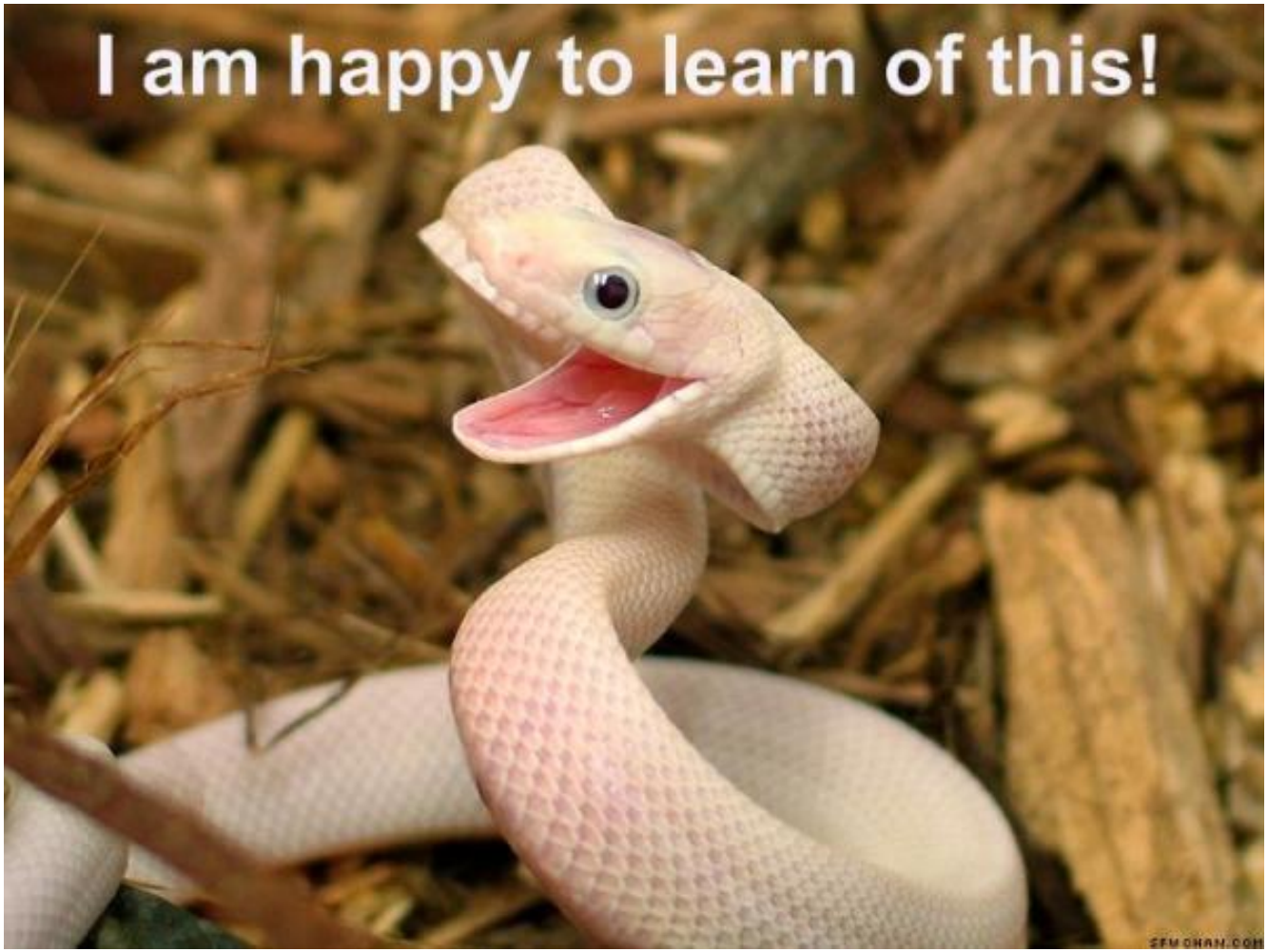
Ideally, we would ...

- use a high level language, to describe an interpreter
- get performance by dynamic compilation
- separate language semantics from design decisions

n*m*l problem

- n - dynamic languages
- m - design decisions (GC, JIT, etc.)
- l - platforms (JVM, .NET, C/Posix)
- we want an `n+m+l` effort, instead of `n*m*l` !

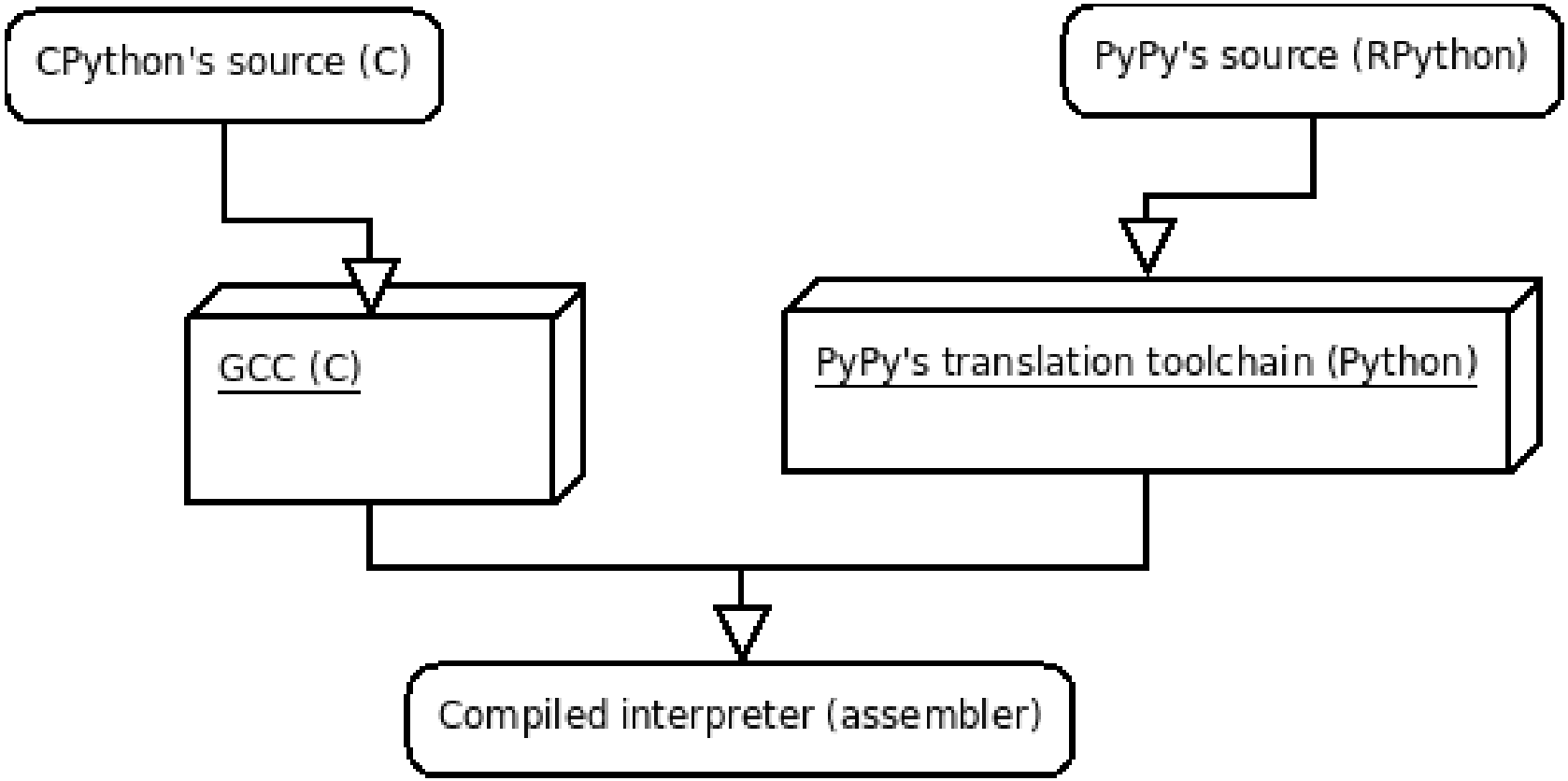
Happy snakes



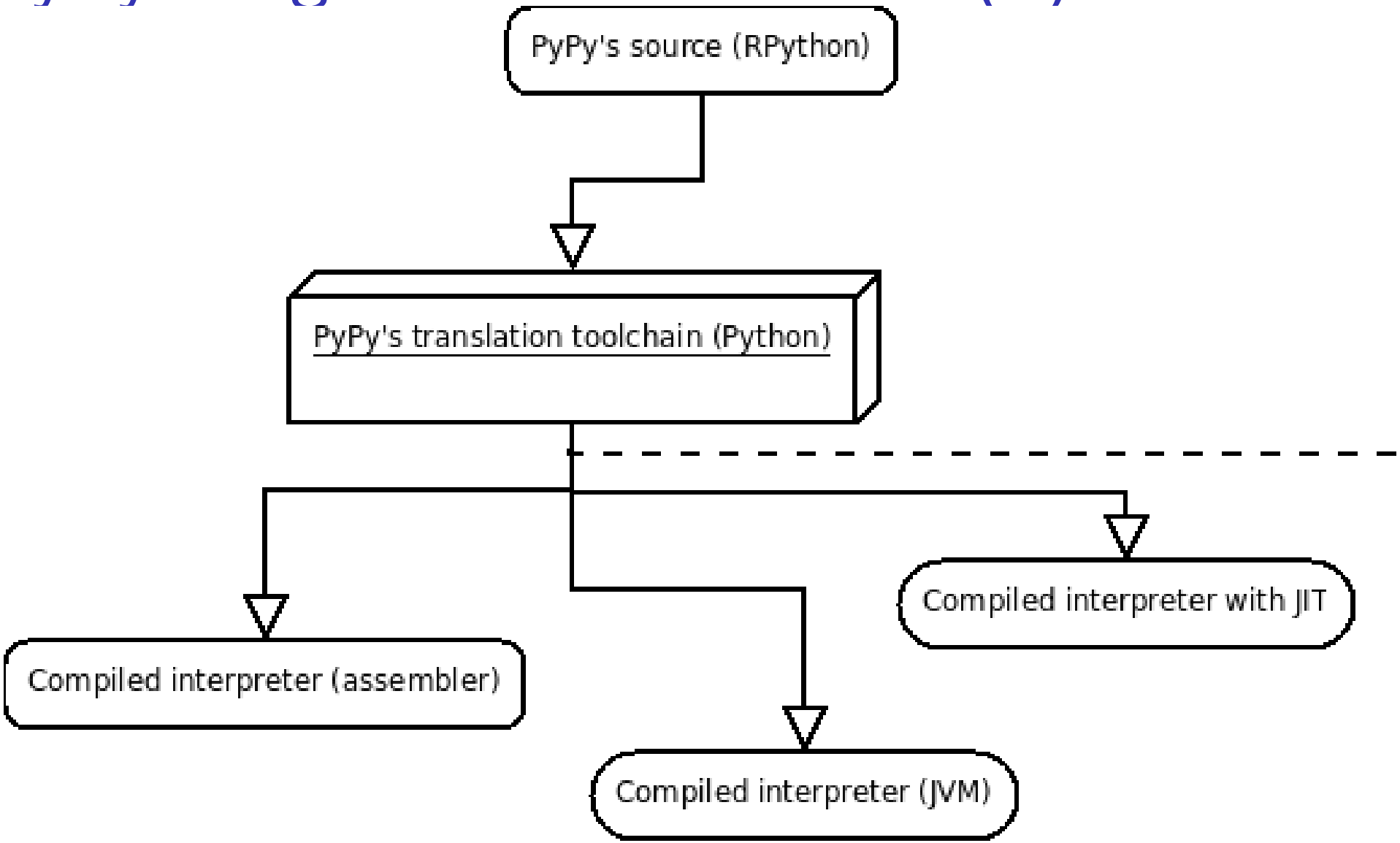
PyPy - high level goals

- solve n*m*l problem
- create a nice, fast and maintainable python implementation
- that runs on C/Posix, .NET, JVM, whatever
- with JIT

PyPy - high level architecture



PyPy - high level architecture (2)



PyPy - implementation language

- RPython - restricted subset of Python
- but still valid python
- static enough to compile to efficient code
- static enough to be able to analyze it
- not necessarily a nice language
- ... but better than C

An example of benefit

- Python has complicated semantics
- Python guarantees that it won't segfault on a stack exhaustion
- CPython includes some stack checks in the source, but they don't catch every case
- We include it automatically so all cases are guaranteed to be covered

PyPy - why so complex?

- interpreter is just a relatively small piece of code
- because we control translation toolchain, we can have work done for us
- nice example - garbage collection
- hopefully even nicer example - JIT

Python interpreter status

- almost as fast as CPython (from 0.8x to 4x slower)
- it's 99% compatible with CPython
- comes with majority of CPython's stdlib (including ctypes)
- able to run django, twisted, pylons...

PyPy - garbage collection

- written in RPython
- completely separated from interpreter
- nicely testable on top of Python
- faster than CPython ones
- very easy to write new ones

JIT - why?

- dynamic compilation is a way to go for dynamic languages
- examples: psyco, strongtalk, modern JavaScript engines
- python is a very complex language, far more complex than JavaScript
- hence, JIT generator is required!

JIT - status

- 4th generation of JIT in PyPy
- 1st generation was able to speed up examples up to 60x (same as psyco)
- psyco (0th generation) was able to speedup python programs up to the same as gcc -O0
- not ready yet
- based on tracing JIT by Michael Franz (the same paper as tracemonkey is based on)
- written in RPython!

JIT - the main idea

- we find hotspot loops, which are executed often enough
- we "trace" those loops and produce machine code
- next time we enter the loop, we execute fast machine code
- we do this on interpreter source level, instead of doing that on python's source code

JIT - a zoom into tracing

- first, hot spot is found
- then, it's traced by a jit to find out what operations really occured
- a loop is closed by a jump
- assembler code from list of operations performed is generated

Project future

- more JIT
- more funding
- 1.1 release this spring
- sprint in Wroclaw 7-14 February

Q&A

- this slides: http://codespeak.net/svn/pypy/extradoc/talk/wroclaw2009/talk.pdf
- blog: http://morepypy.blogspot.com
- project webpage: http://codespeak.net/pypy