PyPy - The new Python implementation on the block

Author: Holger Krekel & Carl Friedrich Bolz
Date: 28th December 2005
Location: 22C3, Berlin

1 The missing talker: Armin Rigo

2 Python implementation facts

- Parser/Compiler produces bytecode
- Virtual Machine interprets bytecode
- strongly dynamically typed
- clean object model at Python and C level

3 Python implementations

- CPython: main Python version (BDFL'ed by Guido)

- Jython: compiles to Java Bytecode
- IronPython (MS): compiles to .NET's CLR
- PyPy: self-contained - self-translating - flexible

4 PyPy project facts

- started 2003 as a grass-root effort
- aims: flexibility, research, speed
- test-driven development
- received EU-funding from end 2004 on
- 350 subscribers to pypy-dev, 150.000 LOCs, 20.000 visitors per month,
- MIT license

5 PyPy development method

- sprints
- test-driven
- open source culture
- see talk tomorrow 2pm (29th Dec. 2005)

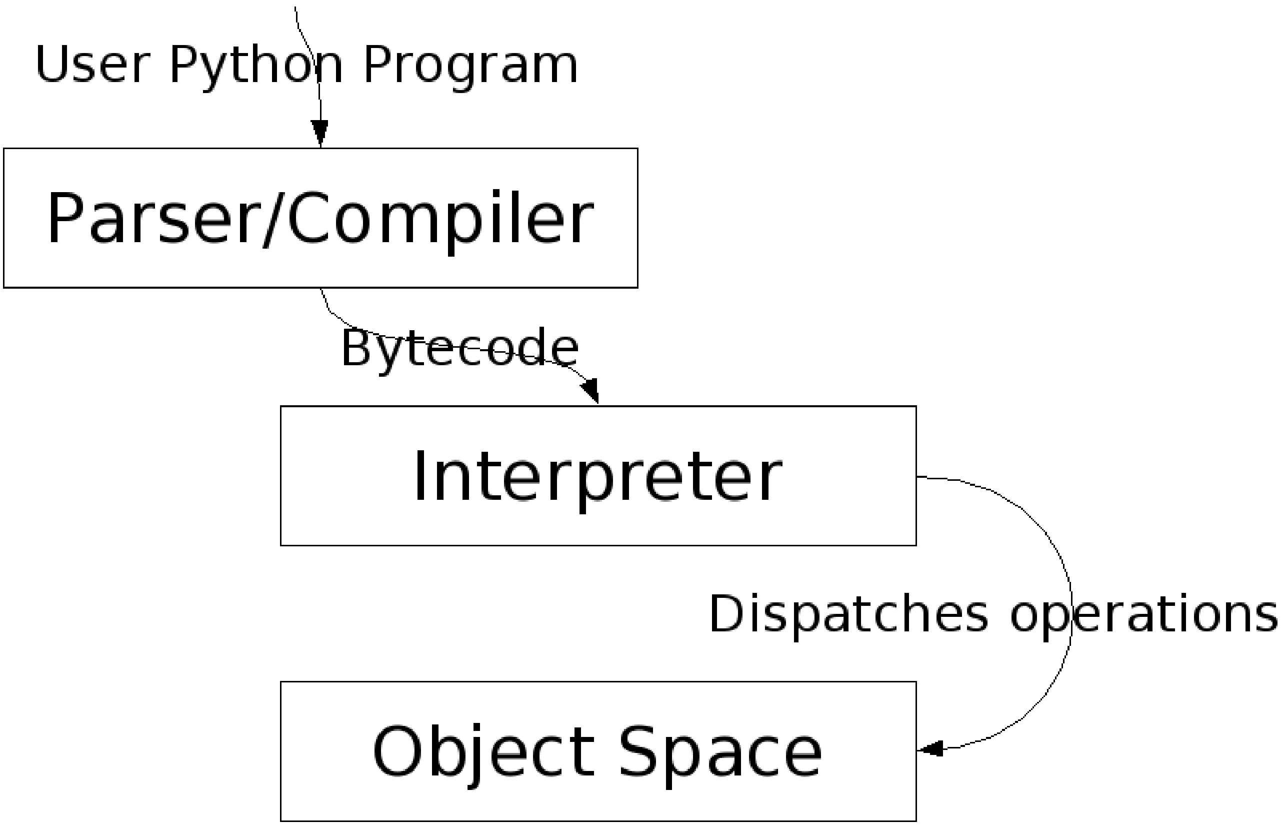
6 PyPy implementation facts

- implements Python language in Python itself
- parts implemented in a restricted subset: RPython
- "static enough" for full-program type inference
- at boot time we allow unrestricted python!

7 PyPy/Python architecture

- parser and compiler
- bytecode interpreter
- Standard Object Space / Type implementations
- Python VM = interpreter + Standard Object Space
- builtin and fundamental modules

8 PyPy/Python architecture picture



9 Parser and Compiler

- parses python source code to AST
- compiles AST to code objects (bytecode)
- works from the CPython grammar definition
- can be modified/extended at runtime (almost)
- (interactive command line dis-example) ...

10 Bytecode interpreter

- interprets bytecode/code objects through Frame objects
- Frames tie to global and local variable scopes
- implements control flow (loops, branches, exceptions, calls)
- dispatches all operations on objects to an Object Library or "Object Space"

11 Object Spaces

- library of all python types and operations on them
- encapsulates all knowledge about app-level objects
- is not concerned with control flow or bytecode
- e.g. enough control to implement lazy evaluation

12 Builtin and Fundamental Modules

- around 200 builtin functions and classes
- fundamental modules like ’sys’ and ’os’ implemented
- quite fully compliant to CPython’s regression tests
- a number of modules missing or incomplete (socket ...)

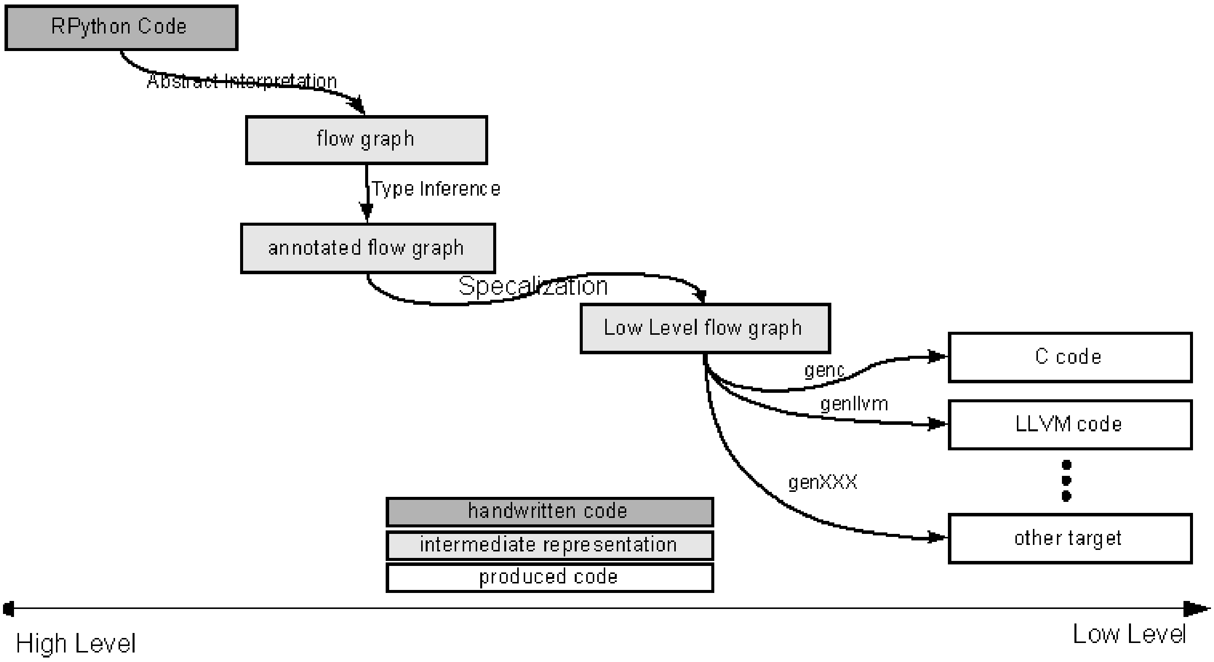
13 Animation on Interpreter/Objspace interaction

- shown on pygame-window ...

14 PyPy/Translation architecture

- bytecode interpreter
- Abstract Interpretation (Flow Object Space)
- Type Inference (Annotation)
- Specialising to lltypesystem / ootypesystem
- C and LLVM Backends to lltypesystem

15 PyPy/Translation overview



16 Abstract Interpretation

- bytecode interpreter dispatches to Flow Object Space
- Flow Object Space implements abstract operations
- produces flow graphs as a side effect
- starts from “live” byte code NOT source code
- pygame demonstration

17 Type Inference

- performs forward propagating type inference
- is used to infer the types in flow graphs
- needs types of the entry point function’s arguments
- assumes that the used types are static
- goes from very special to more general values

18 Specialization

- annotated flow graphs are specialized for language families
- lltypesystem (for C like languages): C, LLVM
- ootypesystem (for OO languages): Java, Javascript, Smalltalk
- result is specialized flow graphs
- these contain operations at target level

19 Backends

- produce code out of specialized flow graphs
- complete backends: C, LLVM
- ongoing: JavaScript, Squeak
- foreign function calls: manually written glue snippets
- big example

20 Translation Aspects

- implementation decisions (GC, threading, CC) at translation time
- most other language implementations do a "fixed" decision
- translation aspects are weaved into the produced code
- independent from language semantics (python interpreter)

21 Aspects: Memory Models

- Currently implemented: refcounting, Boehm-collector
- **more general exact GCs (not yet integrated)**
  - copying
  - mark & sweep
  - ...
- different allocation strategies - not yet

22 Aspects: Threading Models

- currently implemented: single thread and global interpreter lock
- future plans: free threading models
- stacklessness: don't use the C stack for user-level recursion
- Continuation Passing Style (CPS)
- implemented as a part of the backends

23 comparison to other approaches

Project	languages	environments	impl aspects
PyPy	1 (for now)	variable	variable
JVM/Java	variable	1	semi-variable
.NET	variable	1	semi-variable

- environments: language backends, standard runtime environments
- implementation aspects: GC, threading, calling conventions, security, ...

24 three public releases

- 0.6 quite compliant python implementation
- 0.7 compliant self-contained python implementation
- 0.8 full parser and compiler, "10-50 times" better speed

25 lots of documentation

- http://codespeak.net/pypy
- 23rd December: release of 10 PyPy reports to the EU
- talks, papers, slides available on the site

26 PyPy cross pollination

- perl6: Object Spaces
- llvm
- cpython

- squeak (started last CCC conf)
- IronPython/Microsoft

27 one thing: the speed issue

- currently interpreting programs 5-15 times slower than CPython
- now seriously starting with optimisations at various levels
- pypy can translate (R-)python code to something 10-100 times faster compared to running on top of CPython

28 technical outlook 2006

- specialising JIT-compiler, processor backends
- stackless/non-C calling conventions (CPS)
- GC / threading integration + extensions
- orthogonal persistence and distribution (see thunk example)
- built-in security (e-lang ...)

29 outlook on whole project level

- surviving the EU review in Bruxelles 20th January 2006
- improve interactions with community & contribution
- taking care about post-EU development (2007++)
- visiting the US, Japan ...
- commercial opportunities ...

http://codespeak.net/pypy