PyPy and The Art of Generating Virtual Machines

Antonio Cuni
DISI, Università degli Studi di Genova

PyCon Due 2008 - Firenze

May 10, 2008

PyPy

PyPy is both:

- *a Python interpreter written in Python*
- *a framework for writing dynamic languages*

Today we will focus on the latter.

Agenda

- quick overview & motivations
- the translation framework
- JIT generator
- (demo)

PyPy status update (1)

- JVM backend completed, pypy-jvm
- some new GCs, much faster than ever
- ctypes for PyPy
- JIT refactoring, needed to make the JIT production-ready
- improved .NET integration for pypy-cli
- new blog: http://morepypy.blogspot.com

PyPy status update (2)

- various performance improvements
- slighly slower than CPython on pystone (10-20%)
- but faster on richards (20-24%)
- less than 2x slower on other benchmarks

Interpreters

...are good to implement dynamic languages:

- Easy to write
- Portable
- Flexible and easy to evolve, if written in high-level language (without low-level details)

Folk Wisdom

...about interpreters for Dynamic Languages:

- There are unavoidable tradeoffs between flexibility, maintainability, and speed
- Fast, Maintainable, Flexible -- pick one

What this means in Practice

Current popular open source dynamic language implementations:

- are relatively slow
- are not very flexible
- are harder to maintain than we would like them to be

Not very flexible

- Low-level decisions permeate the entire code base.
- Not ideal to experiment - cannot simply plug-in a new garbage collector, memory model, or threading model
- Early decisions come back to haunt you.

Hard to maintain

because they are traditionally written in low-level languages:

- the community generates experts in the dynamic language but requires experts in C or C++ for its own maintenance
- every time a new VM is needed, the language's community forks (CPython - Jython - IronPython)

The PyPy Project

We built enough infrastructure such that:

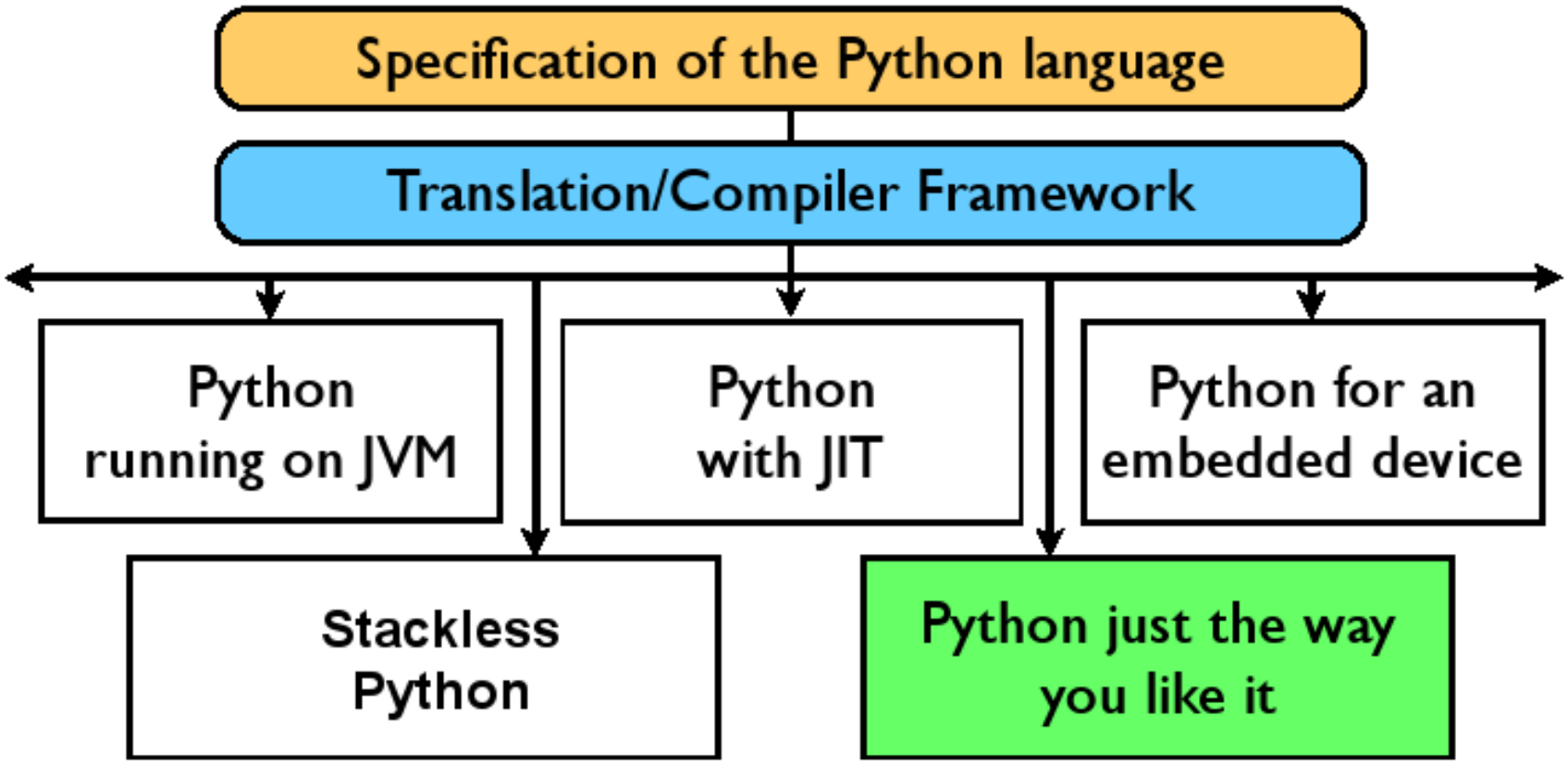
- speed is regained
- features requiring low-level manipulations are (re-)added as *aspects*
- interpreters are kept simple and uncluttered

Targets as different as C and the industry OO VMs (JVM, CLR) are supported.
A special aspect: **Generating JIT compilers**

PyPy as a project

- We operate both as an open source with production usage aspirations and research project.
- We focus on the whole system.
- We want the tool-chain itself to be as simple as possible (but not simpler).
- Some of what we do is relatively straight-forward, some is challenging (generating dynamic compilers!).

PyPy Approach



Going from interpreters to VMs

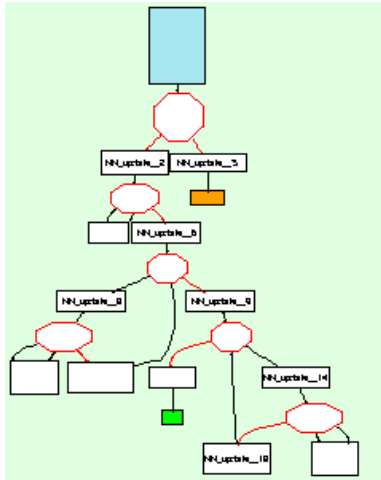
In PyPy interpreters are written in RPython:

- A subset of Python amenable to static analysis
- Still fully garbage collected
- Rich built-in types

RPython is still close to Python.

Translation details (1)

- First, load and initialize RPython code inside a normal Python VM
- RPython translation starts from the resulting "live" bytecode
- Unified "intermediate code" representation: a forest of *Control Flow Graphs*



Translation details (2)

PyPy uses abstract interpretation extensively:

- to construct Flow Graphs
- for type inference
- to gather info for some optimisations
- for Partial Evaluation in the generated Dynamic Compilers...

also uses Flow Graph transformation and rewriting.

Type Systems (1)

We model the different targets through different type systems:

- LL (low-level C-like targets): data and function pointers, structures, arrays...
- OO (object oriented targets): classes and instances with inheritance and dispatching

Type Systems (2)

Translation:

- starts from *RPython Flow Graphs*
- turns them into *LL Flow Graphs* or *OO Flow Graphs*
- the flowgraphs are transformed in various ways
- then they are sent to the backends.

Translation aspects (1)

The interpreters in RPython are free of low-level details (as required to target platforms as different as Posix/C and the JVM/.NET).

- Advanced features related to execution should not need wide-spread changes to the interpreters
- Instead, the interpreters should use support from the translation framework

Translation aspects (2)

Examples:

- GC and memory management
- memory layout
- stack inspection and manipulation
- unboxed integers as tagged pointers

Implementation

- Translation aspects are implemented as transformation of low-level graphs
- Calls to library/helper code can be inserted too
- The helper code is also written in RPython and analyzed and translated

GC Framework

The LL Type System is extended with allocation and address manipulation primitives, used to express GC in RPython directly.

- GCs are linked by substituting memory allocation operations with calls into them
- Transformation inserts bookkeeping code, e.g. to keep track of roots
- Inline fast paths of allocation and barriers

JIT motivation

Flexibility vs. Performance:

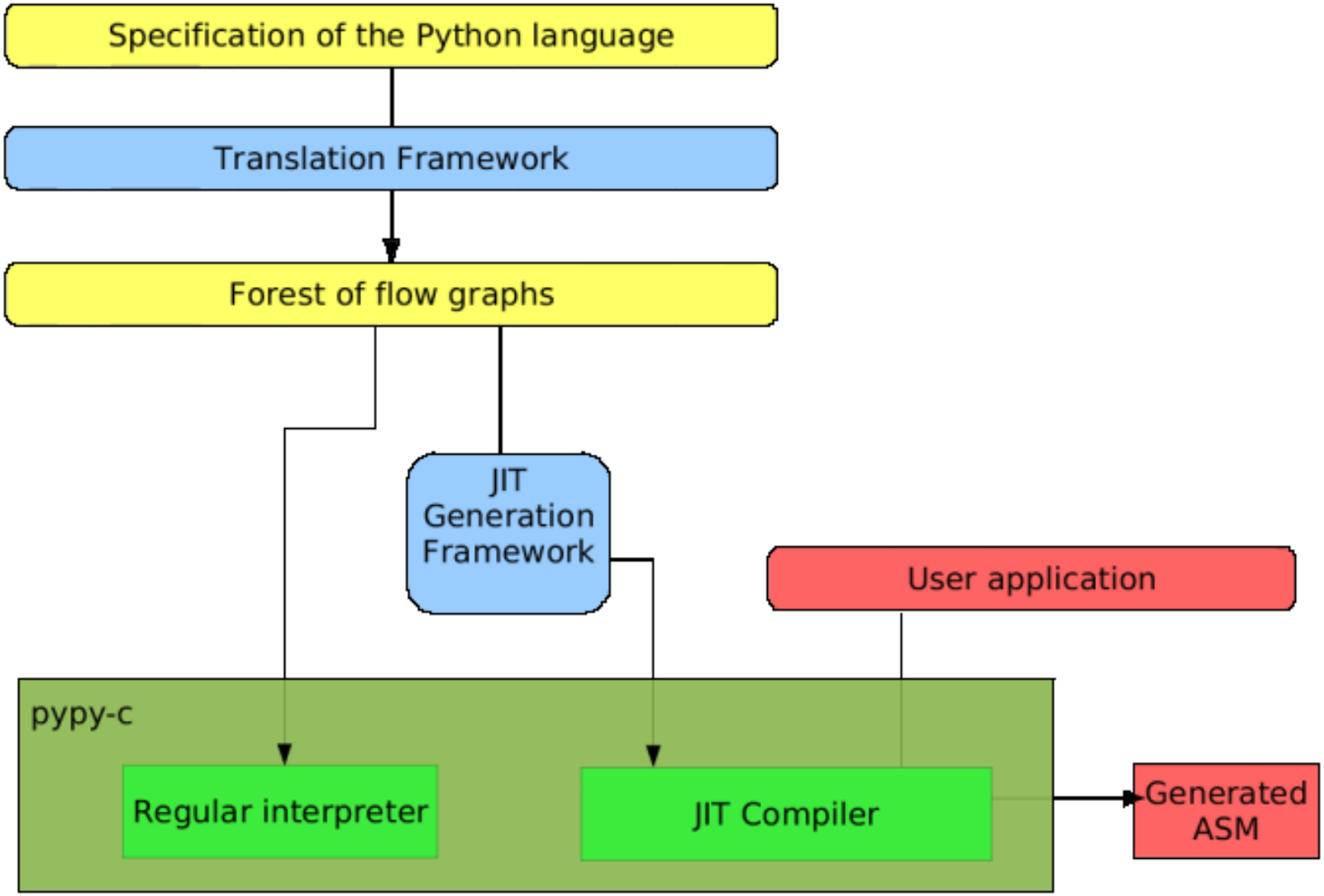
- Interpreters are easy to write and evolve
- For high performance, dynamic compilation is required

Traditional JIT compilers

- Huge resource investment
- The richer the semantics, the harder to write
- Poor encoding of language semantics
- Hard to evolve

Need for novel approaches!

PyPy Architecture



Basics

- Use partial evaluation techniques to generate a dynamic compiler from an interpreter
- Inspiration: Psyco
- Our translation tool-chain was designed for trying this

Futamura

- *Partial evalution of computation process - an approach to a compiler-compiler*, 1971
- Generating compilers from interpreters with automatic specialization
- Relatively little practical impact so far

General idea

Partial evaluation (PE):

- Assume the Python bytecode to be constant, and constant-propagate it into the Python interpreter.

PE for dummies

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=3

```
def f_3(y):
  y2 = y * y
  return 9 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

PE for dummies

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=3

```
def f_3(y):
  y2 = y * y
  return 9 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

PE for dummies

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=3

```
def f_3(y):
  y2 = y * y
  return 9 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

Challenges

- A shortcoming of PE is that in many cases not much can be really assumed constant at compile-time: poor results
- Effective dynamic compilation requires feedback of runtime information into compile-time
- For a dynamic language: types are a primary example

Solution: Promotion

- Enhance PE with the ability to "promote" run-time values to compile-time
- Leverage the dynamic setting

Overall ingredients

The pieces to enable effective dynamic compiler generation in PyPy:

- a few hints in the Python interpreter to guide the JIT generator
- *promotion*
- lazy allocation of objects (only on escape)
- use CPU stack and registers for the contents of the Python frame

Language-agnostic

- The dynamic generation process and primitives are language-agnostic.
- The language implementations should be able to evolve up to maintaining the hints.
- By construction all interpreter/language features are supported

pypy-c-jit

PyPy 1.0 contains both the dynamic compiler generator and the start of its application to PyPy's Python intepreter.
JIT refactoring in-progress.

- included are backends for IA32 and PPC
- experimental/incomplete CLI backend
- integer arithmetic operations are optimized
- for these, we are in the speed range of `gcc -O0`

EXTRA MATERIAL

- More about the JIT Generation:
  - The *Rainbow interpreter*
  - *Virtuals* and *Promotion*

Execution steps

- Translation time
  - pypy-c-jit is translated into an executable
  - the JIT compiler is automatically generated
- Compile-time: the JIT compiler runs
- Runtime: the JIT compiled code runs
- Compile-time and runtime are intermixed

The Rainbow interpreter

- A special interpreter whose goal is to produce executable code
- Written in RPython
- Guided by a binding time analysis ("color" of the graphs)
- Green operations: executed at compile-time
- Red operations: produce code that executes the operation at runtime

Rainbow architecture

Translation time

- Low-level flowgraphs are produced
- The *hint-annotator* colors the variables
- The *rainbow codewriter* translates flowgraphs into rainbow bytecode

Compile-time

- The rainbow interpreter executes the bytecode
- As a result, it procude executable code

Runtime

- The produced code is executed

Rainbow architecture

Translation time

- Low-level flowgraphs are produced
- The *hint-annotator* colors the variables
- The *rainbow codewriter* translates flowgraphs into rainbow bytecode

Compile-time

- The rainbow interpreter executes the bytecode
- As a result, it procude executable code

Runtime

- The produced code is executed

Rainbow architecture

Translation time

- Low-level flowgraphs are produced
- The *hint-annotator* colors the variables
- The *rainbow codewriter* translates flowgraphs into rainbow bytecode

Compile-time

- The rainbow interpreter executes the bytecode
- As a result, it procude executable code

Runtime

- The produced code is executed

Coloring

- Green: compile-time value
- Red: runtime value
- The hints give constraints from which the colors of all values are derived

We reuse the type inference framework to propagate colors

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
    x2 = x * x
    y2 = y * y
    return x2 + y2
```

case x=10

```
def f_10(y):
    y2 = y * y
    return 100 + y2
```

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
    x2 = x * x
    y2 = y * y
    return x2 + y2
```

case x=10

```
def f_10(y):
    y2 = y * y
    return 100 + y2
```

Partial Evaluate Control Flow

- red split points: schedule multiple compilation states
- merge points: merge logic to reuse code for equivalent states

Example

```
if x:
  print "x is true"
if y:
  print "y is true"
```

case y != 0

```
if x:
  print "x is true"
print "y is true"
```

Partial Evaluate Control Flow

- red split points: schedule multiple compilation states
- merge points: merge logic to reuse code for equivalent states

Example

```
if x:
  print "x is true"
if y:
  print "y is true"
```

case y != 0

```
if x:
  print "x is true"
print "y is true"
```

Partial Evaluate Control Flow

- red split points: schedule multiple compilation states
- merge points: merge logic to reuse code for equivalent states

Example

```
if x:
  print "x is true"
if y:
  print "y is true"
```

case y != 0

```
if x:
  print "x is true"
print "y is true"
```

Promotion

Promotion is implemented generating a switch that grows to cover the seen runtime values

- First compilation stops at a promotion point and generates a switch with only a default case. The default will call back into the compiler with runtime values.
- On callback the compiler adds one more case to the switch and generate more code assuming the received value.

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

Virtuals + Promotion

Example from PyPy (simplified!)

```
def add_python_objects(obj1, obj2):
  obj1cls = hint(obj1.__class__, promote=True)
  obj2cls = hint(obj2.__class__, promote=True)
  if obj1cls is IntObject and obj2cls is IntObject:
    x = obj1.intval
    y = obj2.intval
    z = x + y
    return IntObject(intval=z)
```

Conclusion (JIT)

- Effective dynamic compiler generation
  - flexibility and ease of evolution
  - **orthogonal to the performance question**.
- Languages implemented as **understandable interpreters**.
- PyPy proves this a viable approach worth of further exploration.

Open Issues

- inlining control
- promotion switch explosion fallbacks
- jit only the hot-spots
- more hints needed in PyPy's Python
- JIT backends for CLI/JVM

Virtualizable Frames

- frames need to live in the heap (tracebacks ...) and be introspectable
- jit code wants local variables to live in registers and on the stack
- => mark the frame class as "virtualizable"
- jit code uses lazy allocation and stores some contents (local variables...) in register and stack
- outside world access gets intercepted to be able to force lazy virtual data into the heap