The PyPy Interpreter Framework

Chris Narrikkattu
Roadmap

• Common issues in implementing dynamic language VMs
• PyPy Interpreter Framework as a solution
• JIT Generation via the Framework
• Downsides of the Framework
The Issues of Dynamic Language Implementations

• Dynamic language implementations are implemented/tied to specific platforms (CPython vs. Jython vs. IronPython)

• Encodes low-level details and design decisions directly into implementation (clutter)

• Adding a global implementation feature such as a JIT is difficult (see Unladen Swallow, Psyco, Stackless Python), experiments are costly!
The Idea of the PyPy Framework

• Write your interpreter in a high-level language
• Translate your interpreter to add in platform-specific features/decisions
• RPython is a subset of Python amenable to flow and type analysis, thus PyPy could create a Python interpreter entirely in Python!
Framework Architecture

From "PyPy's Approach to Virtual Machine Construction",
Armin Rigo and Samuele Pedroni
Live code objects, not an AST!

Allows for arbitrary (normal) Python as a metaprogramming language while importing.
Framework Architecture

From "PyPy's Approach to Virtual Machine Construction", Armin Rigo and Samuele Pedroni
Object Space Architecture

- Bytecode Interpreter for flow control
- Object Space implements operations on objects
- Flow Object Space records flow graph
- Reuse Bytecode Interpreter in flow graphing

![Diagram](image.png)
Framework Architecture

input RPython program

CPython Interpreter

function objects

Flow Graphing

Type Inference

forest of typed flow graphs

Transformation 1

Transformation 2

Transformation 3

Back-end

compiled code

Make concrete lower-level decisions (how to implement GC? Transform entire interpreter to have a JIT?)
Framework Architecture

input RPython program

CPython Interpreter → function objects

Flow Graphing → Type Inference

forest of typed flow graphs

Transformation 1
Transformation 2
Transformation 3
Back-end

compiled code

Generate code for target platform (C vs. JVM vs. CLR, etc.)
JIT Compiler Generation

• Writing a decent JIT for your new language manually is hard (ask the JS implementers..)
• Keeping semantics the same between non-JIT and JIT is a source of bugs
• Changing your language is harder because you have to change the JIT as well
• PyPy, in contrast, lets you **generate** a JIT automatically for your language
If you write your interpreter in Lua and run with LuaJIT

- **Input**: MyLang Code
- **Lua interpreting Implementation**: LuaJIT compiles the Lua to machine code
- **Implementation interpreting MyLang Bytecode**: LuaJIT DOES NOT optimize your language (the MyLang code)

**Output**
If you write your interpreter in RPython and translate it to C with the JIT option set:

MyLang Code → Implementation as machine code → JIT Tracer

Translation turned your interpreter into C

JIT understands MyLang, emits specialized code for MyLang program
JIT Compiler Generation

• Generates a tracing JIT (trace loops in code)
• JIT is of your interpreter specialized to the user's code, not just the interpreter itself
• Based on partial evaluation techniques
JIT Compiler Generation

• Automatically generated
  – Added a new bytecode? Just retranslate!

• Small marginal effort to add (a few hints)

```python
from rpython.rlib.jit import JitDriver
jitdriver = JitDriver(greens=['pc', 'program', 'bracket_map'], reds=['tape'])

# top of bytecode loop
while pc < len(program):
    jitdriver.jit_merge_point(pc=pc, tape=tape, program=program,
                               bracket_map=bracket_map)

def jitpolicy(driver):
    from rpython.jit.codewriter.policy import JitPolicy
    return JitPolicy()
```
"What RPython allows one to do is profoundly different to the traditional route. **In essence, one writes an interpreter and gets a JIT for free.** I suggest rereading that sentence again: it **fundamentally changes the economics** of language implementation for many of us."

- Laurence Tratt,"Fast Enough VMs in Fast Enough Time"
Other PyPy Interpreter Flexibility

• Thunk Object Space
• Taint Object Space
• GC choice (Boehm, generational, etc.)
• Stackless transformation
• Implementation optimizations (such as tagged pointer for small ints)
Downsides

• RPython is not Python, you do not have full dynamism

• Translation framework is long and memory intensive (45 mins/2-4 GB for PyPy Python interpreter)

• Full retranslate on any change
  – You can do tests without translating by running as (slow) normal Python program
Summary

• Creating a language VM is hard enough work
• RPython is high-level vehicle to allow translation through a framework
• PyPy Framework allows transforms such as adding a JIT compiler with very little marginal cost to implementer
• Concentrate on semantics while getting acceptable speed for almost free
More Resources

- Main PyPy Site
  http://pypy.org
- Papers and Talks
- RPython
- “Fast Enough VMs in Fast Enough Time”
  http://tratt.net/laurie/tech_articles/articles/fast_enough_vms_in_fast_enough_time
- “Tutorial: Writing an Interpreter with PyPy, Part 1”