PyPy – a progress report

ACCU 2006/Python UK, Oxford

Michael Hudson mwh@python.net
Heinrich-Heine-Universität Düsseldorf
What is PyPy?

- PyPy is:
  - An implementation of Python
  - A very flexible compiler framework
  - An open source project (MIT license)
  - A STREP (“Specific Targeted REsearch Project”), partially funded by the EU
  - A lot of fun!
• We can currently produce a binary that looks very much like CPython to the user

• It’s fairly slow (around the same speed as Jython)

• Can also produce binaries that are more capable than CPython – with stackless-style coroutines, with logic variables, ...
Motivation

• PyPy grew out of a desire to modify/extend the implementation of Python, for example to:
  • increase performance (psyco-style JIT compilation, better garbage collectors)
  • add expressiveness (stackless-style coroutines, logic programming)
  • ease porting (to new platforms like the JVM or CLI or to low memory situations)
Lofty goals, but first...

- 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)
Enter the PyPy platform

Specification of the Python language

Compiler Framework

Python running on JVM

Python with JIT

Python for an embedded device

Python with transactional memory

Python just the way you like it
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 lets us write unit tests for our specification/implementation that run on top of CPython

• Can also test entire specification/implementation in same way
The “What is RPython?” question

• Restricted Python, or RPython, first and foremost it is Python

• It is a subset of Python that is static enough – after initialization code has run – for our analysis tools to cope with

• Somewhat Java-like – classes, methods, no pointers, no operator overloading
The “What is RPython?” question

• The definition of RPython is basically “what our compiler can analyze” – so changes (slowly) as toolchain does

• The property of “being RPython” belongs to entire programs and not, say, functions or modules because the annotator performs a global analysis
Translation Aspects

- Our Python implementation is very high level
- One of our Big Goals is to produce our customized Python implementations without compromising on this point
- We do this by weaving in so-called ‘translation aspects’ during the compilation process
In more detail…

Standard Interpreter
- Bytecode Evaluator
- Standard Object Space
- Parser/Compiler

Compiler Framework
- Flow Analysis
- Annotator
- RTyper
- Backend

written in RPython
written in full Python
The standard interpreter does roughly speaking the same job as CPython does, and is split into three chunks.

CPython can be split along the same lines with enough imagination – hardly a coincidence!
The Standard Interpreter

Bytecode Evaluator

Standard Object Space

Parser/Compiler

The bytecode evaluator evaluates the same bytecodes as CPython but treats objects as black boxes – it doesn’t care if they are Python-like values, abstract Variables or even fruit.

\[ 2 + 3 = 5 \]

Variable + Constant = Variable

\[ \text{banana} + \text{orange} = \text{lemon} \]
The Standard Interpreter

The Standard Object Space implements objects that look very much like CPython’s – integers, lists, dictionaries, classes, etc. (it’s a bit different on the inside though)
The Standard Interpreter

The parser and compiler, perhaps predictably, parses Python code and compiles it – to the same bytecode as CPython uses.

Will sometime soon allow runtime modification of the grammar of the language.
The standard interpreter is pretty stable now, implementing Python 2.4.3 (and some 2.5 features),

Some work to come on the parser/compiler and logic variable integration.
Compiler Framework

Flow Analysis
Annotator
RTyper
Backend
Compiler Framework

Flow Analysis

Annotator

RTyper

Backend

Analyzes a single code object to deduce control flow

We have a funky pygame flow graph viewer that we use to view these flow graphs
Compiler Framework

- Flow Analysis
- Annotator
- RTyper
- Backend

Analyzes an entire program to deduce type and other information.

Uses abstract interpretation, rescheduling and other funky stuff.
Compiler Framework

- Flow Analysis
- Annotator
- RTyper
- Backend

Uses the information found by the annotator to decide how to lay out the types used by the input program in memory, and translates high level operations to lower level more pointer-ish operations.
Compiler Framework

Flow Analysis
Annotator
RTyper
Backend

Translates low level operations and types from the RTyper to (currently) C, JavaScript or LLVM code

Sounds like it should be easy, in fact a bit painful
The Flow Model

- Without going into details of how the Flow Object Space works, it produces a control flow graph of a code object.

- Values are either Variables or Constants.

- Operations are described by SpaceOperations like "add".

- SpaceOperations live in Blocks which are connected by Links.
The Flow Model

- **SpaceOperations** have an **opname**, a **result variable** and a list of **args**

- **A few examples:**
  - \( c = a + b \rightarrow v_c = \text{add}(v_a, v_b) \)
  - \( z = x.y \rightarrow v_z = \text{getattr}(v_x, \text{“}y\text{”}) \)
  - \( t = f(u) \rightarrow v_t = \text{simple_call}((f), v_u) \)

- **Demo!**
The Annotator

• Type annotation is a fairly widely known concept – it associates variables with information about which values they might take at run time

• An unusual feature of PyPy’s approach is that the annotator works on live objects

• This means it never sees initialization code, so that can use `exec` and other insane tricks
The Annotator

- Works by abstractly interpreting the control flow graphs produced by the flow analysis
- Annotation starts at a given entry point and discovers as it proceeds which functions may be called by the input program
- Read “Compiling dynamic language implementations” on the web site for more than is on these slides
The Annotator

- Does not modify the graphs; end result is essentially a big dictionary mapping Variables to instances of a subclass of SomeObject

- Important subclasses are SomeInteger, SomeList, SomeInstance, SomePBC ("some pre-built constant", includes classes and functions)

- Demo
The RTyper

• RTyper takes as input an annotated RPython program (e.g. our Python implementation)

• Performs “representation selection” and converts high-level operations to low-level

• Potentially can target a C-ish, pointer-using language or an OO language like Java or Smalltalk with classes and instances (OO backend not yet complete)
For example, in:

```python
for x in range(10):
    print x
```

we represent the `range` as just start/stop/step.

But in:

```python
l = range(10)
l[2] = 3
```

`l` is represented as a full (mutable) list.
Instances of the class \texttt{LowLevelType} describe a C-like types – structures or arrays, pointers or primitive types such as integer or float.

The RTyper attaches a \texttt{LowLevelType} to each Variable and Constant in each annotated control flow graph.
Many high level operations apply to different types; for example you can "add" strings, floats or integers and continually having to distinguish is annoying.

Easier for backends to have monomorphic operations `int_add`, `float_add`, `str_add`.

Some operations are more complex, e.g. instantiation of a class.
Translating High Level to Low Level

- We saw that the code “z = x + y” becomes “v_z = add(v_x, v_y)”.

- Assuming that v_x and v_y (and thus v_z) are annotated as SomeInteger, then:
  - v_x, v_y, v_z will get a LowLevelType of Signed
  - the "add" operation will be replaced with an "int_add" operation
The Backend(s)

- Maintained backends: C, JavaScript and LLVM (Smalltalk and CLI/.NET on the way)
- All proceed in two phases:
  - Traverse the forest of rtyped graphs, computing names for everything
  - Spit out the code
Status – what works

• The Standard Interpreter very nearly complete

• The compiler framework:
  • Produces standalone binaries
  • C and LLVM backends well supported
  • JavaScript backend works, but not for all of PyPy
Status – what works

• The C backend supports three garbage collection strategies:
  • reference counting,
  • using the conservative Boehm-Demers-Weiser collector
  • a precise mark and sweep collector we wrote
Status – what works

- The C and JavaScript backends support “stackless” features – coroutines, tasklets, recursion only limited by RAM
- Can use OS threads with a simple “GIL-thread” model
- Our Python specification/implementation has remained free of all these implementation decisions!
What we’re working on now

• The Just-In-Time compiler – early stages, works for a very simple language

• More home-grown GCs (e.g. a semispace copying collector) and more GCs for LLVM

• Logic programming – some working code, interface and integration in progress
What we’re working on now

- “rctypes”, a uniform way of calling external functions based on the now-standard “ctypes” module for CPython
- CLI (.NET) and Smalltalk backends
- supporting stackless features in other backends
About the project

• Open source, of course (MIT license)
• Distributed – the 12 paid developers live in 6 countries, contributors from more
• Sprint driven development – focused week long coding sessions every ~6 weeks
• Extreme Programming practices: pair programming, test-driven development
“We’re Hiring!”

• In the open source sense:
  • Read documentation:
    http://codespeak.net/pypy/
  • Come hang out in #pypy on freenode, post to pypy-dev
  • Some opportunities for paid work too.