PyPy – where we are now

(So no talk about our plans to conquer the world)

PyCon 2006, Dallas, Texas

Michael Hudson mwh@python.net
Heinrich-Heine-Univeristät Düsseldorf

What is PyPy?

- PyPy is:
  - An implementation of Python, written in Python
  - An open source project
  - A STREP ("Structred Targeted REsearch Proposal"), partially funded by the EU
  - A lot of fun!

Demo

- 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 -- stackless, thunk, ...
- Talk is mainly about how we got here

Overview

- PyPy can be has two major components:
  - The “standard interpreter” which implements the evaluator loop and object semantics of CPython
  - The analysis tool chain that can analyze the standard interpreter and, for example, compile it to C.

The Standard Interpreter

- Written in RPython, but runs on CPython too (RPython is approximately defined as "what the analysis tool chain accepts" – see next slide)
- Consists of a parser/compiler, a bytecode interpreter and a Standard Object Space
- Functionally equivalent to CPython

The "What is RPython?" question

- Restricted Python, or RPython for short, is a subset of Python that is static enough for our analysis toolchain to cope with
- First and foremost it *is* Python -- this lets us unit test our standard interpreter
- Definition not fixed -- changes as toolchain does

The Interpreter/ Object Space split

- The byte code interpreter treats objects as black boxes and consistently references a “space” object to manipulate them
- This allows us to use a funky object space to help analysis (later)
- The Standard Object Space implements objects that look very much like CPython’s

The Parser/ Compiler

- The standard interpreter includes a parser and bytecode compiler for Python
- Based on work by Jonathan David Riehl and CPython's compiler package (but with less bugs)
- Allows/will allow runtime modification of syntax and grammar

The Analysis Tool Chain

- Has four main parts:
  - The Flow Object Space
  - The Annotator
  - The RTyper
  - The Low Level backend
- Quick tour coming up, then more about RTyper

Flow Object Space

- Technically speaking, an "abstract domain" for the bytecode interpreter
- Treats objects as either "Constants" or "Variables"
- Works on a code object at a time
- Produces a control flow graph
- Basically stable since early 2005

The Annotator

- The RPythonAnnotator analyzes an entire RPython program to infer types and inter-function control flow
- Interesting stuff, but well documented -- see "Compiling dynamic language implementations" on the website for more.
- More-or-less stable since early summer 2005

The RTyper

- First of all: "the RTyper" is badly named
- Converts the still-fairly-high-level output of the annotator into lower level concepts that are easier to translate into languages like C
- More after the next slide
- Basically working since summer 2005

The Low-Level Backend(s)

- Take the low-level operations produced by the RTyper and converts to a low-level language
- At time of writing, C and LLVM are the supported targets
- Working though not stable from spring 2005

Why talk about the RTyper?

- We haven't talked about it much before at conferences :)
- The need for it took us slightly by surprise
- It doesn't fit in the standard Computer Science syllabus
- It's interesting

What does it do?

- In some sense the annotator observes ("abstractly interprets") the input program and records what it sees
- The RTyper makes decisions based on these records -- working out how instances of a given class should be laid out in memory for example
- Takes advantage of global analysis

What does it *actually* do?

- The RTyper does two main things:
  - decide how each object in the input program will be represented in terms of a C-like model (so, structs, arrays, integers, function pointers…)
  - replace the operations in the control flow graphs with operations that manipulate these low level representations

What does the RTyper work on?

The output of the annotator, which for this snippet:

```
x = y.z
```

might be something like

```
v_x <- getattr(v_y, "z")
```

with *annotations*:

```
{v_y: an instance of some class "Y",
 v_x: an integer}
```

What does the RTyper produce?

- Assuming the only attribute to ever be seen on an instance of Y is "z", instances of Y will probably look something like this:

```
GcStruct("Y", ("inst_z", Signed))
```

- The getattr operation will be replaced by

```
v_x <- getfield(v_y, "inst_z")
```

- This might be translated to C as:

```
v_x = v_y->inst_z;
```

A little about the project

- Open Source, of course (MIT license)
- About 12 people work on PyPy full time
- Distributed -- full timers live in six(?) countries
- Welcoming -- watch out, you might get hired! (easier for Europeans, admittedly)
- Sprinting after the conference

Current work

- Writing a Just In Time compiler
- Integration of logic programming/constraint solving
- Modularizing the garbage collection system
- Generally refactoring – a noticeable trend has been to do less and at source generation time