PyPy Project

• You've probably heard about PyPy
• Python implemented in Python
• It is apparently quite fast
• Smart people seem to work on it
• How it works: Magic? Souls of grad students?
Concerns

- Honestly, PyPy scares me a little bit
- Actually, it's the implementation that scares me
- Can I make it fit my brain?
- Can "normal" programmers understand it?
- Can you debug it?
- Can you modify it?
Building PyPy from source (at 64x speed)

On a machine with 8GB RAM

Please Explain (and debug)
Premise

• I think a big part of Python's success is due to the fact that "normal" programmers are easily able to tinker with the implementation

• Written in ANSI C

• Uses standard tools (make, autoconf, etc.)

• Well documented
High-level Docs

Welcome! This is the documentation for Python 2.7.2, last updated Jan 11, 2012.

Parts of the documentation:

- What's new in Python 2.7?
  or all "What's new" documents since 2.0
- Tutorial
  start here
- Library Reference
  keep this under your pillow
- Language Reference
  describes syntax and language elements
- Python Setup and Usage
  how to use Python on different platforms
- Python HOWTOs
  in-depth documents on specific topics
- Extending and Embedding
  tutorial for C/C++ programmers
- Python/C API
  reference for C/C++ programmers
- Installing Python Modules
  information for installers & sys-admins
- Distributing Python Modules
  sharing modules with others
- Documenting Python
  guide for documentation authors
- FAQs
  frequently asked questions (with answers!)
Allocating Objects on the Heap — Python v2.7.2 documentation

PyObject* _PyObject_New(PyTypeObject *type)


PyVarObject* _PyObject_NewVar(PyTypeObject *type, Py_ssize_t size)

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

void _PyObject_Del(PyObject *op)

PyObject* PyObject_Init(PyObject *op, PyTypeObject *type)

Return value: Borrowed reference.
Initialize a newly-allocated object op with its type and initial reference. Returns the initialized object. If type indicates that the object participates in the cyclic garbage detector, it is added to the detector's set of observed objects. Other fields of the object are not affected.

PyVarObject* PyObject_InitVar(PyVarObject *op, PyTypeObject *type, Py_ssize_t size)

Return value: Borrowed reference.
This does everything PyObject_Init() does, and also initializes the length information for a variable-size object.

Changed in version 2.5: This function used an int type for size. This might require changes in your code for properly supporting 64-bit systems.

TYPE* PyObject_New(TYPE, PyTypeObject *type)
Why it Matters

- People can submit bug reports (with patches)
- People can make extensions
- People can port Python to new environments
- People can experiment with it
- Everything great about Python has happened because of tinkering
PyPy

- An advanced research project
- Lots of academic papers and tech reports
- Many high-level presentations
- A fair bit of documentation
- A lot of information giving you the "gist"
High-level Docs
Basically, each operation in the flow graphs of the user program generates one such rule. The rules are conditional on the annotations bound to the operation’s input argument variables, in a way that mimics the ad-hoc polymorphic nature of most Python operations. We will not give all rules in the sequel, but focus on representative examples. An `add` operation generates the following rules (where `x`, `y` and `z` are replaced by the variables that really appear in each particular `add` operation in the flow graphs of the user program):

\[
\begin{align*}
  z &= \text{add}(x, y), \ b(x) = \text{Int}, \ \text{Bool} \leq b(y) \leq \text{Int} \\
  b' &= b \text{ with } (z \rightarrow \text{Int}) \\

  z &= \text{add}(x, y), \ \text{Bool} \leq b(x) \leq \text{Int}, \ b(y) = \text{Int} \\
  b' &= b \text{ with } (z \rightarrow \text{Int}) \\

  z &= \text{add}(x, y), \ \text{Bool} \leq b(x) \leq \text{NonNegInt}, \ \text{Bool} \leq b(y) \leq \text{NonNegInt} \\
  b' &= b \text{ with } (z \rightarrow \text{NonNegInt}) \\

  z &= \text{add}(x, y), \ \text{Char} \leq b(x) \leq \text{NullableStr}, \ \text{Char} \leq b(y) \leq \text{NullableStr} \\
  b' &= b \text{ with } (z \rightarrow \text{Str})
\end{align*}
\]

The rules are read as follows: for the operation \( z = \text{add}(x, y) \), we consider the bindings of the variables `x` and `y` in the current state \((b, E)\); if the bindings satisfy the given conditions, then the rule is applicable. Applying the rule means producing a new state \((b', E')\) derived from the current state -- here by changing the binding of the result variable `z`.

Note that for conciseness we omitted the \( E' = E \) (none of these rules modify `E`).
Detailed Tech Reports

Basically, each operation in the flow graphs of the user program generates one such rule. The rules are conditional on the annotations bound to the operation’s input arguments variables, in a way that mimics the ad-hoc polymorphic nature of most Python code. To give all rules in the sequel, but focus on representative examples. And here are the following rules (where $x$, $y$ and $z$ are replaced by the variables that represent the particular add operation in the flow graphs of the user program):

$$
\begin{align*}
    z &= \text{add}(x, y), \quad b(x) = \text{Int}, \quad \text{Bool} \leq b(y) \leq \text{Int} \\
    b' &= b \text{ with } (z \to \text{Int})
\end{align*}
$$

$$
\begin{align*}
    z &= \text{add}(x, y), \quad \text{Bool} \leq b(x) \leq \text{Int}, \quad b(y) = \text{Int} \\
    b' &= b \text{ with } (z \to \text{Int})
\end{align*}
$$

$$
\begin{align*}
    z &= \text{add}(x, y), \quad \text{Bool} \leq b(x) \leq \text{NonNegInt}, \quad \text{Bool} \leq b(y) \leq \text{NonNegInt} \\
    b' &= b \text{ with } (z \to \text{NonNegInt})
\end{align*}
$$

$$
\begin{align*}
    z &= \text{add}(x, y), \quad \text{Char} \leq b(x) \leq \text{NullableStr}, \quad \text{Char} \leq b(y) \leq \text{NullableStr} \\
    b' &= b \text{ with } (z \to \text{Str})^1
\end{align*}
$$

The rules are read as follows: for the operation $z = \text{add}(x, y)$, we consider the bindings of the variables $x$ and $y$ in the current state $(b, E)$; if the bindings satisfy the given conditions, then the rule is applicable. Applying the rule means producing a new state $(b', E')$ derived from the current state -- here by changing the binding of the result variable $z$.

Note that for conciseness we omitted the $E' = E$ (none of these rules modify $E$).
Basically, each operation in the flow graphs of the user program generates one such rule. The rules are conditional on the annotations bound to the operation's input argument variables, in a way that mimics the ad-hoc polymorphic nature of most Python, but focus on representative examples. And the following rules (where \( x, y \) and \( z \) are replaced by the variables of particular add operation in the flow graphs of the user program):

\[
\begin{align*}
\frac{z = \text{add}(x, y), \ b(x) = \text{Int}, \ \text{Bool} \leq b(y) \leq \text{Int}}{b' = b \text{ with } (z \to \text{Int})}
\end{align*}
\]

\[
\begin{align*}
\frac{z = \text{add}(x, y), \ \text{Bool} \leq b(x) \leq \text{Int}, \ b(y) = \text{Int}}{b' = b \text{ with } (z \to \text{Int})}
\end{align*}
\]

\[
\begin{align*}
\frac{z = \text{add}(x, y), \ \text{Bool} \leq b(x) \leq \text{NonNegInt}, \ \text{Bool} \leq b(y) \leq \text{NonNegInt}}{b' = b \text{ with } (z \to \text{NonNegInt})}
\end{align*}
\]

\[
\begin{align*}
\frac{z = \text{add}(x, y), \ \text{Char} \leq b(x) \leq \text{NullableStr}, \ \text{Char} \leq b(y) \leq \text{NullableStr}}{b' = b \text{ with } (z \to \text{Str})}
\end{align*}
\]

The rules are read as follows: for the operation \( z = \text{add}(x, y) \), we consider the bindings of the variables \( x \) and \( y \) in the current state \((b, E)\): if the bindings satisfy the given conditions, then the rule is applicable. Applying the rule means producing a new state \((b', E')\) derived from the current state -- here by changing the binding of the result variable \( z \).

Note that for conciseness we omitted the \( E' = E \) (none of these rules modify \( E \)).

To be fair, it's a funded academic project in PL. They have no other choice than to write like this.

(maybe I'll just read the source code)
This Talk

• A tiny bit about how PyPy works at an implementation level (e.g., code)

• Specifically, rpython

• Based on a lot of personal tinkering with it

• Mainly, I'm just curious

• Is there anything to take away?
Disclaimer

• I am not affiliated with PyPy in any way
• Have not used it for any real project
• Have contributed nothing to it except a bug report about bad GIL behavior (sic)
• Have general awareness based on various conference presentations, blog posts, etc.
PyPy Overview

- PyPy is Python implemented in Python

- You can run PyPy as a normal Python script
Running `py.py`

- Running as a script...

```
bash % python py.py
[platform:execute] gcc-4.0 -c -arch x86_64 -O3 -fomit-frame-pointer - \\ 
[platform:execute] gcc-4.0 -c -arch x86_64 -O3 -fomit-frame-pointer - \\ ...
PyPy 1.7.0 in StdObjSpace on top of Python 2.7.2 (startuptime: 34.51 secs)
```

- Performance is dreadful
- Just for testing
**rpython**

- PyPy is actually implemented in "rpython"
- rpython is not an "interpreter", but a restricted subset of the Python language
- It can run as valid Python code, but that's about the only similarity
rpython

- rpython is a completely different language
  - Python syntax, yes.
  - Must be compiled (like C, C++, etc.)
  - Static typing via type inference
- If you love Python, you will hate rpython
- Closest comparable language I've used: ML
Hello World

- **Sample rpython Program**

  ```python
  # hello.py

  def main(argv):
      print "Hello World"
      return 0

  def target(*args):
      return main, None
  ```

- **Must have a C-like entry point (main)**

- **Must define target() to identify the entry**
Translation (Compilation)

• rpython programs must be translated

```
bash % pypy/translator/goal/translate.py hello.py

[platform:msg] Setting platform to 'host' cc=None
[translation:info] Translating target as defined by hello
[platform:execute] gcc-4.0 -c -arch x86_64 -O3 -fomit-frame-pointer -mdynamic-no-pic /var/folders/... lots of additional output ...
```

• Creates a C program and compiles it

```
bash % ./hello-c
Hello World
bash %
```
A Real World Example

• Fibonacci numbers (of course)

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```
A Real World Example

- Fibonacci numbers (of course)

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

Yes, it is fast
Type Inference

- Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```
Type Inference

- Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

```python
int
```
Type Inference

- Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

```python
int
```
Type Inference

• Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

```python
int
```

```python
int
```
Type Inference

- Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

```python
int
```

```python
int
```
Type Inference

• Type inference illustrated

```python
# fib.py

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def main(argv):
    print fib(int(argv[1]))
    return 0

def target(*args):
    return main, None
```

It's fast because types are attached to everything (like C)

Resulting code is stripped of all "dynamic" features
R is for Restricted

- rpython allows no dynamic typing

```python
def add(x,y):
    return x+y

def main(argv):
    r1 = add(2,3)              # Ok
    r2 = add("Hello","World")  # Error
    return 0
```

- Functions can only have one type signature
- Determined on first use
Sample Error Message

raise AnnotatorError(msgstr)
AnnotatorError': annotation of 'union' degenerated to SomeObject()
Simple call of incompatible family:
(KeyError getting at the binding!)

In <FunctionGraph of (func:4)main at 0x181c7d8>:
Happened at file func.py line 6

r1 = add(2,3)
==>

r2 = add("Hello","World")

Previous annotation:
 (none)

.. v8 = simple_call((function add), ('Hello'), ('World'))
.. '(func:4)main'

Processing block:
block@9 is a <class 'pypy.objspace.flow.flowcontext.SpamBlock'>
in (func:4)main
containing the following operations:

v3 = simple_call((function add), (2), (3))
v8 = simple_call((function add), ('Hello'), ('World'))
R is for Restricted

• Containers can only have a single type

```python
numbers = [1, 2, 3, 4, 5]               # Ok
items   = [1, "Hello", 3.5]         # Error

names   = {                           # Ok
    'dabeaz' : 'David Beazley',
    'gaynor' : 'Alex Gaynor',
}

record = {                          # Error
    'name' : 'ACME',
    'shares' : 100
}
```

• Think C, not Python.
R is for Restricted

• Attributes can only be a single type

```python
class Pair(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

a = Pair(2, 3)              # OK (first use)
b = Pair("Hello", "World")  # Error
```

• Again, think C
R is for Restricted

• Mixing datatypes requires boxing/unboxing

```python
class SomeValue(object):
    pass

class IntValue(SomeValue):
    def __init__(self, value):
        self.value = value
    def getint(self):
        return self.value

class StrValue(SomeValue):
    def __init__(self, value):
        self.value = value
    def getstr(self):
        return self.value

# Error
record = {
    'name' : 'Dave',
    'clout' : 13
}

# OK
record = {
    'name' : StrValue('Dave'),
    'clout' : IntValue(13)
}

print record['name'].getstr()
print record['clout'].getint()
```

• All objects are of type "SomeValue"
R is for Restricted

- PyPy developers seem to indicate that end-users shouldn't mess around with rpython
- I agree
- It's not the python that you know
- Trades speed for annoyance
- Missing a lot of features (e.g., generators)
The really interesting part of rpython is the translation process.

rpython takes your Python program and turns it into C code which is then compiled.

This is done without "parsing" your program or doing anything that looks like the operation of traditional compiler.
rpython Translation

- Translation process works on a live imported version of your code in a standard Python interpreter
- Driven entirely through introspection of the underlying bytecode
- Let's peel back the covers....
# Some Python code

def ctest(a, b, c):
    d = a + b
    if d < c:
        e = d - c
    else:
        e = d + c
    return e

>>> dis.dis(ctest)
    4          0 LOAD_FAST                0 (a)              0 (a)
    3          3 LOAD_FAST                1 (b)              1 (b)
    6     >>  6 BINARY_ADD
    7          7 STORE_FAST               3 (d)              3 (d)
    5          10 LOAD_FAST               3 (d)              3 (d)
    13         13 LOAD_FAST               2 (c)              2 (c)
    16         16 COMPARE_OP               0 (<)
    19         19 JUMP_IF_FALSE           14 (to 36)
    22         22 POP_TOP                  
    6          23 LOAD_FAST               3 (d)              3 (d)
    26         26 LOAD_FAST               2 (c)              2 (c)
    29         29 BINARY_SUBTRACT
    30         30 STORE_FAST               4 (e)              4 (e)
    33         33 JUMP_FORWARD            11 (to 47)
    >> 36     >> 36 POP_TOP                  
    8          37 LOAD_FAST               3 (d)              3 (d)
    40         40 LOAD_FAST               2 (c)              2 (c)
    43         43 BINARY_ADD               
    44         44 STORE_FAST               4 (e)              4 (e)
    9     >> 47 LOAD_FAST               4 (e)
    50         50 RETURN_VALUE

• All Python code is compiled to bytecode
rpython Translation

• Compiled code held in code objects

```python
>>> ctest.__code__
<code object ctest at 0x33a968, file "ctest.py", line 3>
>>> ctest.__code__.co_code
'|\x00\x00|\x01\x00|\x17}\x03\x00|\x03\x00|\x02\x00j\x00|\x00o\n|\x03\x00|\x04\x00n|\x07|\x00|\x01|\x02|\x00|\x04|\x00S'
>>> ctest.__code__.co_varnames
('a', 'b', 'c', 'd', 'e')
>>> ctest.__code__.co_argcount
3
>>> ctest.__code__.co_nlocals
5
>>> 
```

• rpython operates entirely from this (not source)!
Bytecode Interpretation

• A core part of PyPy consists of a Python bytecode interpreter (remember, it's Python implemented in Python)

• A modular design that allows different backends (object spaces) to be plugged into it
Abstract Interpretation

• rpython takes Python code objects from CPython and interprets them using the pypy byte code interpreter (head explodes)

• A special "flow space" monitors and records the actual operations that get performed

• Assembles the operations into a flow graph describing the program
Abstract Interpretation

Instruction stream is "executed" in the abstract

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Offset</th>
<th>Input Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD_FAST</td>
<td>0</td>
<td>(a)</td>
</tr>
<tr>
<td>3</td>
<td>LOAD_FAST</td>
<td>1</td>
<td>(b)</td>
</tr>
<tr>
<td>6</td>
<td>BINARY_ADD</td>
<td>2</td>
<td>(c)</td>
</tr>
<tr>
<td>7</td>
<td>STORE_FAST</td>
<td>3</td>
<td>(d)</td>
</tr>
<tr>
<td>10</td>
<td>LOAD_FAST</td>
<td>4</td>
<td>(d)</td>
</tr>
<tr>
<td>13</td>
<td>LOAD_FAST</td>
<td>5</td>
<td>(c)</td>
</tr>
<tr>
<td>16</td>
<td>COMPARE_OP</td>
<td>6</td>
<td>(&lt;)</td>
</tr>
<tr>
<td>19</td>
<td>JUMP_IF_FALSE</td>
<td>7</td>
<td>(to 36)</td>
</tr>
<tr>
<td>22</td>
<td>POP_TOP</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>LOAD_FAST</td>
<td>9</td>
<td>(d)</td>
</tr>
<tr>
<td>26</td>
<td>LOAD_FAST</td>
<td>10</td>
<td>(c)</td>
</tr>
<tr>
<td>29</td>
<td>BINARY_SUBTRACT</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>STORE_FAST</td>
<td>12</td>
<td>(e)</td>
</tr>
<tr>
<td>33</td>
<td>JUMP_FORWARD</td>
<td>13</td>
<td>(to 47)</td>
</tr>
<tr>
<td>36</td>
<td>POP_TOP</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>LOAD_FAST</td>
<td>15</td>
<td>(d)</td>
</tr>
<tr>
<td>40</td>
<td>LOAD_FAST</td>
<td>16</td>
<td>(c)</td>
</tr>
<tr>
<td>43</td>
<td>BINARY_ADD</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>STORE_FAST</td>
<td>18</td>
<td>(e)</td>
</tr>
<tr>
<td>47</td>
<td>LOAD_FAST</td>
<td>19</td>
<td>(e)</td>
</tr>
<tr>
<td>50</td>
<td>RETURN_VALUE</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Interpretation

Inputs: [a_0, b_0, c_0, None, None, None, None, None]

Start block is created.
Inputs are initial stack frame
Abstract Interpretation

Start executing instructions and updating the frame
Abstract Interpretation

```
0  LOAD_FAST  0  (a)
  3  LOAD_FAST  1  (b)
  6  BINARY_ADD
  7  STORE_FAST  3  (d)
 10  LOAD_FAST  3  (d)
 13  LOAD_FAST  2  (c)
 16  COMPARE_OP  0  (<)
 19  JUMP_IF_FALSE  14  (to 36)
 22  POP_TOP
 23  LOAD_FAST  3  (d)
 26  LOAD_FAST  2  (c)
 29  BINARY_SUBTRACT
 30  STORE_FAST  4  (e)
 33  JUMP_FORWARD  11  (to 47)
 36  POP_TOP
 37  LOAD_FAST  3  (d)
 40  LOAD_FAST  2  (c)
 43  BINARY_ADD
 44  STORE_FAST  4  (e)
 47  LOAD_FAST  4  (e)
 50  RETURN_VALUE
```

Inputs: [a_0, b_0, c_0, None, None, None, None]

Start executing instructions and updating the frame
Abstract Interpretation

Inputs: [a_0, b_0, c_0, None, None, None, None]

Notice how the stack is getting updated
(keeps track of where things are)
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBSTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_0, b_0, c_0, None, None, None, None]

Operation causes the creation of a new block

(inputs represent stack state)
Abstract Interpretation

Inputs: [a_0, b_0, c_0, None, None, None, None]

[ [a_0, b_0, c_0, None, None, a_0, None]  
[a_0, b_0, c_0, None, None, a_0, b_0] ]

Inputs: [a_1, b_1, c_1, None, None, v6, v7]

[ [a_1, b_1, c_1, None, None, v8, None]  
[a_1, b_1, c_1, v8, None, None, None] ]

v8 = add(v6, v7)

Keep updating the frame
Abstract Interpretation

Inputs: \([a_0, b_0, c_0, None, None, None, None]\)

Keep updating the frame
Abstract Interpretation

Inputs: [a_0, b_0, c_0, None, None, None, None]

Keep updating the frame

Inputs: [a_1, b_1, c_1, None, None, v6, v7]

v8 = add(v6, v7)

Inputs: [a_1, b_1, c_1, None, None, v8, None]

[...]

Keep updating the frame
Abstract Interpretation

Inputs: [a_0, b_0, c_0, None, None, None, None]

Inputs: [a_1, b_1, c_1, None, None, v6, v7]

v8 = add(v6, v7)

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

v15 = lt(v13, v14)

Operation means a new block
Abstract Interpretation

Critical: Each operation lives in its own block
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

v15=lt(v13, v14)
[a_2, b_2, c_2, d_0, None, v15, None]

Inputs: [a_3, b_3, c_3, d_1, None, v20, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, v21, None]
Abstract Interpretation

Let's talk branches: Must explore both the true/false branches
Abstract Interpretation

```
0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD 3 (d)
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT 4 (e)
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD 4 (e)
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE
```

Inputs: 
- \([a_2, b_2, c_2, d_0, \text{None}, v13, v14]\)
- \([a_2, b_2, c_2, d_0, \text{None}, v15, \text{None}]\)
- \([a_3, b_3, c_3, d_1, \text{None}, v21, \text{None}]\)
- \([a_3, b_3, c_3, d_1, \text{None}, \text{None}, \text{None}]\)
- \([a_3, b_3, c_3, d_1, \text{None}, v21, \text{None}]\)
- \([a_3, b_3, c_3, d_1, \text{None}, v20, \text{None}]\)
- \([a_3, b_3, c_3, d_1, \text{None}, v21, \text{None}]\)

false
Abstract Interpretation

```
0 LOAD_FAST                0 (a)
3 LOAD_FAST                1 (b)
6 BINARY_ADD
7 STORE_FAST               3 (d)
10 LOAD_FAST                3 (d)
13 LOAD_FAST                2 (c)
16 COMPARE_OP               0 (<)
19 JUMP_IF_FALSE           14 (to 36)
22 POP_TOP
23 LOAD_FAST                3 (d)
26 LOAD_FAST                2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST               4 (e)
33 JUMP_FORWARD            11 (to 47)
36 POP_TOP
37 LOAD_FAST                3 (d)
40 LOAD_FAST                2 (c)
43 BINARY_ADD
44 STORE_FAST               4 (e)
47 LOAD_FAST                4 (e)
50 RETURN_VALUE
```

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

\[v15 = \text{lt}(v13, v14)\]
[\[a_2, b_2, c_2, d_0, None, v15, None\]]

Inputs: [a_3, b_3, c_3, d_1, None, v20, None]

\[v21 = \text{is\_true}(v20)\]
[\[a_3, b_3, c_3, d_1, None, v21, None\]]

false

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

\[v21 = \text{is\_true}(v20)\]
[\[a_3, b_3, c_3, d_1, None, v21, None\]]
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

v15=lt(v13, v14)
[a_2, b_2, c_2, d_0, None, v15, None]

Inputs: [a_3, b_3, c_3, d_1, None, v20, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, v21, None]

false

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, v21, None]

[a_3, b_3, c_3, d_1, None, v20, None]

[a_3, b_3, c_3, d_1, None, None, None]

[a_3, b_3, c_3, d_1, None, d_1, None]

[a_3, b_3, c_3, d_1, None, d_1, c_3]
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

v15=lt(v13, v14)
[a_2, b_2, c_2, d_0, None, v15, None]

Inputs: [a_3, b_3, c_3, d_1, None, v20, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, v21, None]

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

true

false

downarrow

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]

[None, None, None]

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]

[None, None, None]

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]

[None, None, None]
Abstract Interpretation

Inputs: \[ [a_2, b_2, c_2, d_0, \text{None}, v_{13}, v_{14}] \]

\[ v_{15} = \text{lt}(v_{13}, v_{14}) \]

Inputs: \[ [a_2, b_2, c_2, d_0, \text{None}, v_{15}, \text{None}] \]

\[ v_{21} = \text{is_true}(v_{20}) \]

Inputs: \[ [a_3, b_3, c_3, d_1, \text{None}, v_{20}, \text{None}] \]

\[ v_{21} = \text{is_true}(v_{20}) \]

Inputs: \[ [a_3, b_3, c_3, d_1, \text{None}, v_{21}, \text{None}] \]

false

false

true

true
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_2, b_2, c_2, d_0, None, v13, v14]

v15=lt(v13, v14)
[a_2, b_2, c_2, d_0, None, v15, None]

Inputs: [a_3, b_3, c_3, d_1, None, v20, None]
v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, v21, None]

true
false

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]
v21=is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

false
true
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21=is_true(v20)

true

[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

false

[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

v28 = add(v26, v27)

[a_4, b_4, c_4, d_2, None, v26, v27]
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21 = is_true(v20)

true

[v21]
[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

false

[v28]
[a_4, b_4, c_4, d_2, None, v26, v27]
[a_4, b_4, c_4, d_2, None, v28, None]
[a_4, b_4, c_4, d_2, v28, None, None]
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

v21 = is_true(v20)
[ [a_3, b_3, c_3, d_1, None, None, None] ]
[ [a_3, b_3, c_3, d_1, None, d_1, None] ]
[ [a_3, b_3, c_3, d_1, None, d_1, c_3 ] ]

true

Inputs: [a_4, b_4, c_4, d_2, None, v26, v27 ]

v28 = add(v26, v27)
[ [a_4, b_4, c_4, d_2, None, v28, None] ]
[ [a_4, b_4, c_4, d_2, v28, None, None] ]
[ [a_4, b_4, c_4, d_2, v28, None, v21] ]

false
Abstract Interpretation

0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_ADD
7 STORE_FAST 3 (d)
10 LOAD_FAST 3 (d)
13 LOAD_FAST 2 (c)
16 COMPARE_OP 0 (<)
19 JUMP_IF_FALSE 14 (to 36)
22 POP_TOP
23 LOAD_FAST 3 (d)
26 LOAD_FAST 2 (c)
29 BINARY_SUBTRACT
30 STORE_FAST 4 (e)
33 JUMP_FORWARD 11 (to 47)
36 POP_TOP
37 LOAD_FAST 3 (d)
40 LOAD_FAST 2 (c)
43 BINARY_ADD
44 STORE_FAST 4 (e)
47 LOAD_FAST 4 (e)
50 RETURN_VALUE

Inputs: [a_3, b_3, c_3, d_1, None, v21, None]

true
v21 = is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

false
v21 = is_true(v20)
[a_3, b_3, c_3, d_1, None, None, None]
[a_3, b_3, c_3, d_1, None, d_1, None]
[a_3, b_3, c_3, d_1, None, d_1, c_3]

v31 = sub(v29, v30)
[a_5, b_5, c_5, d_3, None, v29, v30]
[a_5, b_5, c_5, d_3, None, v31, None]
[a_5, b_5, c_5, d_3, v31, None, None]

v28 = add(v26, v27)
[a_4, b_4, c_4, d_2, None, v26, v27]
[a_4, b_4, c_4, d_2, None, v28, None]
[a_4, b_4, c_4, d_2, v28, None, None]
Abstract Interpretation

Eventually....
Eventually Get a Flow Graph
It Gets Simplified
This is just the first step
Annotation and Discovery

- After flow graph of entry point is created, rpython starts annotating it
- Flow graph is scanned and types are attached
- If new functions are discovered, their flow graphs are created and they are annotated
- This continues recursively, eventually reaching all corners of your program.
My head hurts...
Final Comments

• None really
• Still trying to wrap my brain around some of the later stages of translation (time issue)
• Extremely challenging (maybe I've missed some documentation?)
One Challenge

• Everything is Python
  • PyPy interprets Python
  • PyPy is written in python (rpython)
  • rpython is implemented in Python
  • Parts of rpython use PyPy code
  • Boom!

• Challenging to sort out what you're looking at