PyPy Intro and JIT Frontend

Antonio Cuni

Intel@Bucharest

April 4 2016

About this talk

- What is PyPy? What is RPython?
- Tracing JIT 101
- PyPy JIT frontend and optimizer
  - "how we manage to make things fast"

Part 1

PyPy introduction

What is PyPy?

- For most people, the final product:

```
$ pypy
Python 2.7.10 (173add34cdd2, Mar 15 2016, 23:00:19)
[PyPy 5.1.0-alpha0 with GCC 4.8.4] on linux2
>>>> import test.pystone
>>>> test.pystone.main()
Pystone(1.1) time for 50000 passes = 0.0473992
This machine benchmarks at 1.05487e+06 pystones/second
```

- More in general: a broader project, ecosystem and community

PyPy as a project

- `rpython`: a fancy compiler
  - source code: "statically typed Python with type inference and metaprogramming"
  - fancy features: C-like performance, GC, meta-JIT
  - "like GCC" (it statically produces a binary)
  - you can run RPython programs on top of CPython (veeery slow, for development only)
- `pypy`: a Python interpreter
  - "like CPython", but written in RPython
  - CPython : GCC = PyPy : RPython

PyPy as a project

- `rpython`: a fancy compiler
  - source code: "statically typed Python with type inference and metaprogramming"
  - fancy features: C-like performance, GC, meta-JIT
  - "like GCC" (it statically produces a binary)
  - you can run RPython programs on top of CPython (veeery slow, for development only)

- `pypy`: a Python interpreter
  - "like CPython", but written in RPython
  - CPython : GCC = PyPy : RPython

Important fact

- We **did not** write a JIT compiler for Python
- The "meta JIT" works with all RPython programs
- The "Python JIT" is automatically generated from the interpreter
- Writing an interpreter is vastly easier than a compiler
- Other interpreters: smalltalk, prolog, ruby, php, ...

The final product

- `rpython` + `pypy`: the final binary you download and execute
  - a Python interpreter
  - with a GC
  - with a JIT
  - fast

Part 2

Overview of tracing JITs

Assumptions

- Pareto Principle (80-20 rule)
  - the 20% of the program accounts for the 80% of the runtime
  - **hot-spots**
- Fast Path principle
  - optimize only what is necessary
  - fall back for uncommon cases

- Most of runtime spent in **loops**
- Always the same code paths (likely)

Assumptions

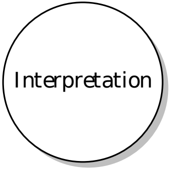
- Pareto Principle (80-20 rule)
  - the 20% of the program accounts for the 80% of the runtime
  - **hot-spots**
- Fast Path principle
  - optimize only what is necessary
  - fall back for uncommon cases

- Most of runtime spent in **loops**
- Always the same code paths (likely)

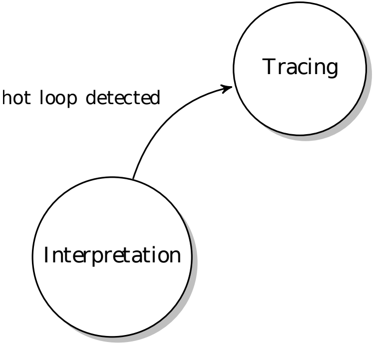
Tracing JIT

- Interpret the program as usual
- Detect **hot** loops
- Tracing phase
  - **linear** trace
- Compiling
- Execute
  - guards to ensure correctness
- Profit :-)

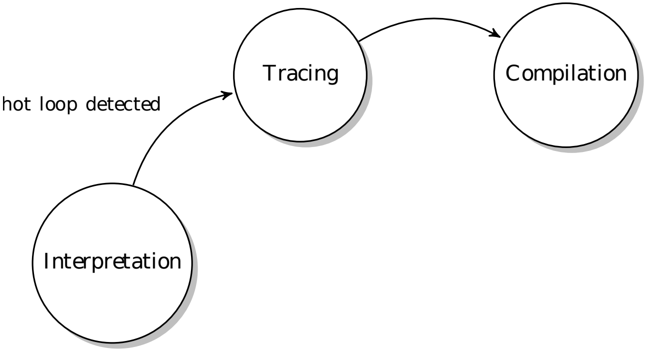
Tracing JIT phases



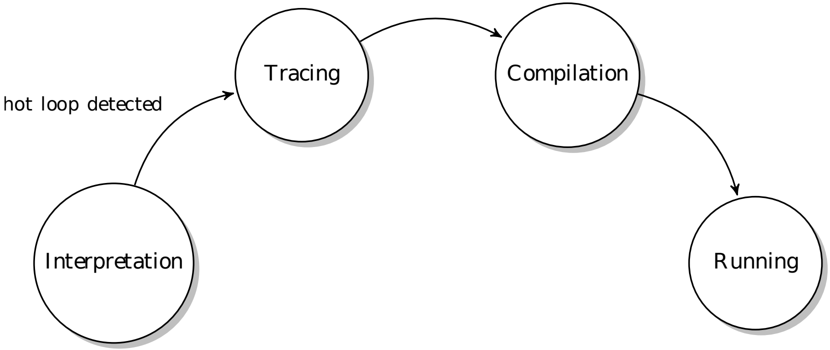
Tracing JIT phases



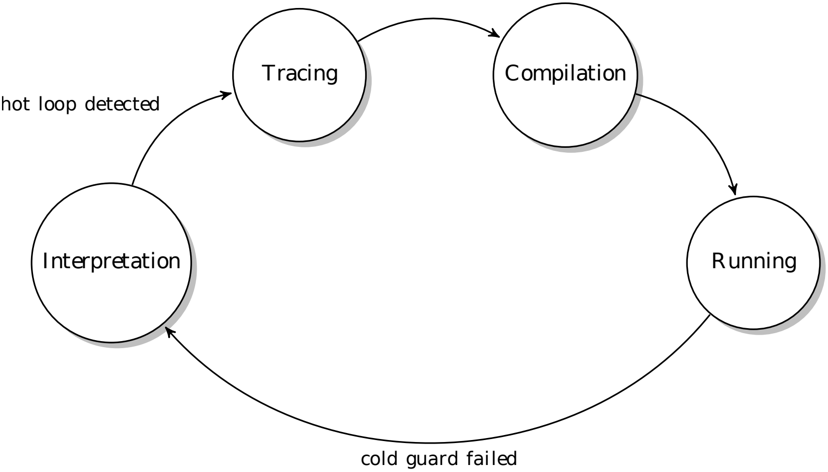
Tracing JIT phases



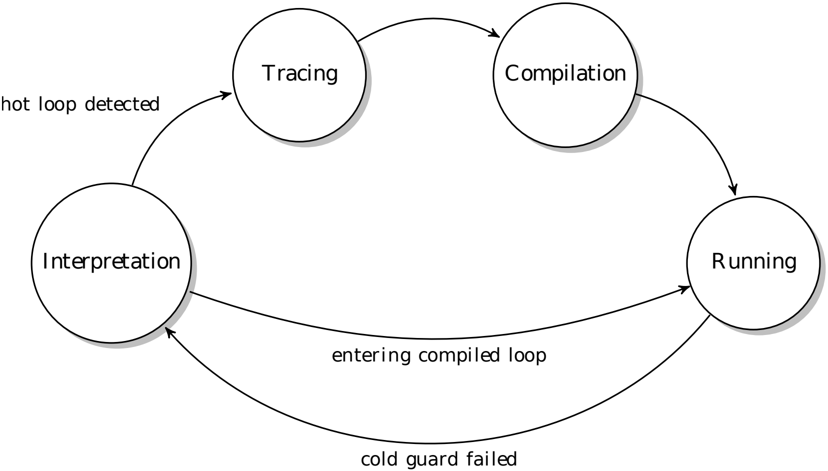
Tracing JIT phases



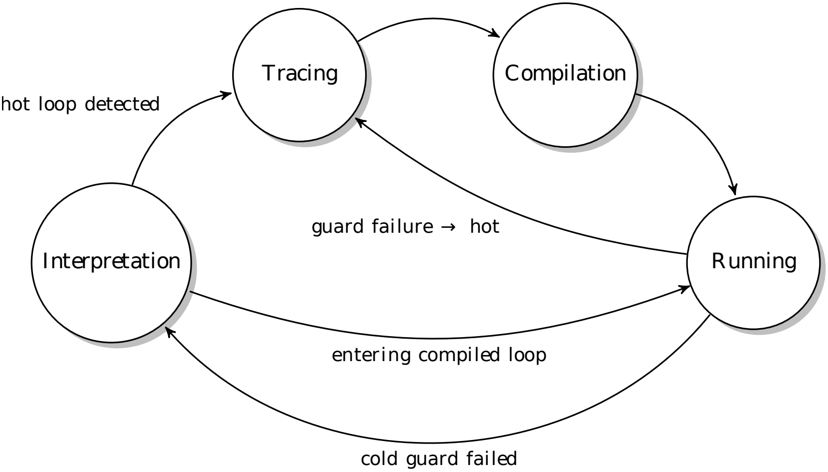
Tracing JIT phases



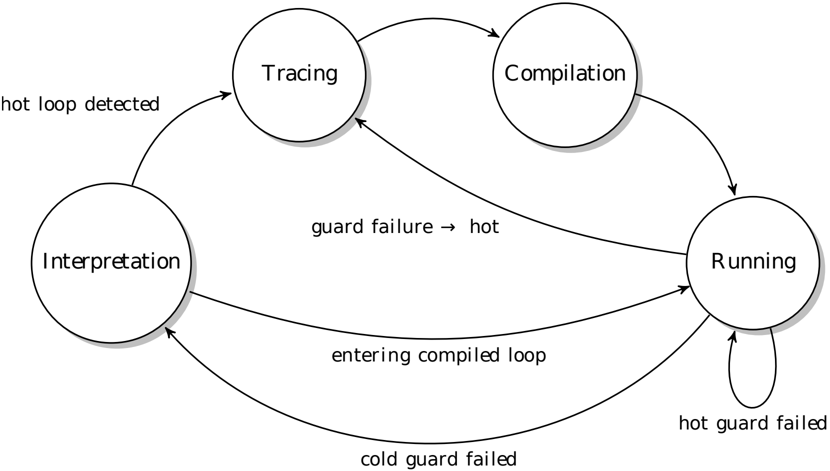
Tracing JIT phases



Tracing JIT phases



Tracing JIT phases



Trace trees

WRITE ME

Part 3

The PyPy JIT

Terminology (1)

- **translation time**: when you run "rpython targetpypy.py" to get the `pypy` binary
- **runtime**: everything which happens after you start `pypy`
- **interpretation**, **tracing**, **compiling**
- **assembler/machine code**: the output of the JIT compiler
- **execution time**: when your Python program is being executed
  - by the interpreter
  - by the machine code

Terminology (2)

- **interp-level**: things written in RPython
- **[PyPy] interpreter**: the RPython program which executes the final Python programs
- **bytecode**: "the output of dis.dis". It is executed by the PyPy interpreter.
- **app-level**: things written in Python, and executed by the PyPy Interpreter

Terminology (3)

- (the following is not 100% accurate but it's enough to understand the general principle)
- **low level op or ResOperation**
  - low-level instructions like "add two integers", "read a field out of a struct", "call this function"
  - (more or less) the same level of C ("portable assembler")
  - knows about GC objects (e.g. you have `getfield_gc` vs `getfield_raw`)
- **jitcodes**: low-level representation of RPython functions
  - sequence of low level ops
  - generated at **translation time**
  - 1 RPython function --> 1 C function --> 1 jitcode

Terminology (4)

- **JIT traces or loops**
  - a very specific sequence of llops as actually executed by your Python program
  - generated at **runtime** (more specifically, during **tracing**)
- **JIT optimizer**: takes JIT traces and emits JIT traces
- **JIT backend**: takes JIT traces and emits machine code

General architecture

RPYTHON

```
def LOAD_GLOBAL(self):
    ...
```

```
def STORE_FAST(self):
    ...
```

```
def BINARY_ADD(self):
    ...
```

General architecture

RPYTHON

```
def LOAD_GLOBAL(self):
    ...
```

```
def STORE_FAST(self):
    ...
```

```
def BINARY_ADD(self):
    ...
```

→ CODEWRITER

General architecture

RPYTHON

```
def LOAD_GLOBAL(self):
    ...
```

```
def STORE_FAST(self):
    ...
```

```
def BINARY_ADD(self):
    ...
```

CODEWRITER

JITCODE

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
....
```

```
...
p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....
```

```
...
promote_class(p0)
i0 = getfield_gc(p0, 'intval')
promote_class(p1)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(i0, i1)
if (overflowed) goto ...
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
....
```

General architecture

RPYTHON

```
def LOAD_GLOBAL(self):
    ...
```

```
def STORE_FAST(self):
    ...
```

```
def BINARY_ADD(self):
    ...
```

CODEWRITER

JITCODE

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
....
```

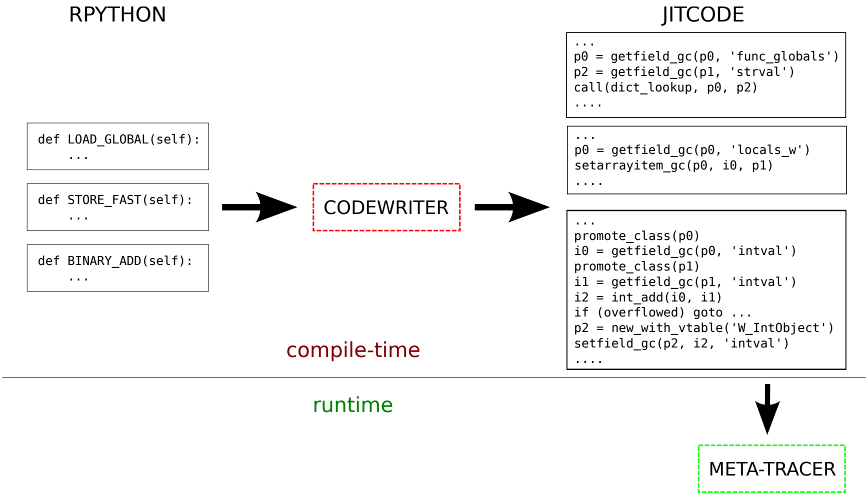
```
...
p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....
```

```
...
promote_class(p0)
i0 = getfield_gc(p0, 'intval')
promote_class(p1)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(i0, i1)
if (overflowed) goto ...
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
....
```

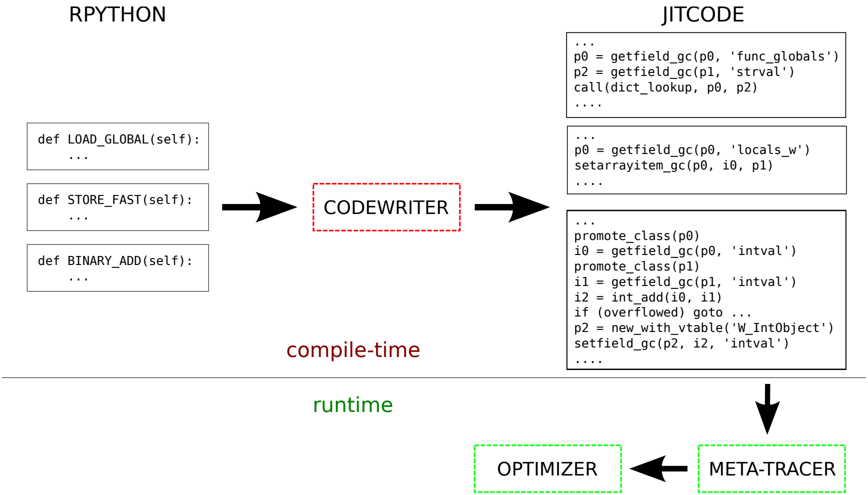
compile-time

runtime

General architecture



General architecture



General architecture

RPYTHON

```
def LOAD_GLOBAL(self):
    ...
```

```
def STORE_FAST(self):
    ...
```

```
def BINARY_ADD(self):
    ...
```

CODEWRITER

JITCODE

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
....
```

```
...
p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....
```

```
...
promote_class(p0)
i0 = getfield_gc(p0, 'intval')
promote_class(p1)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(i0, i1)
if (overflowed) goto ...
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
....
```

compile-time

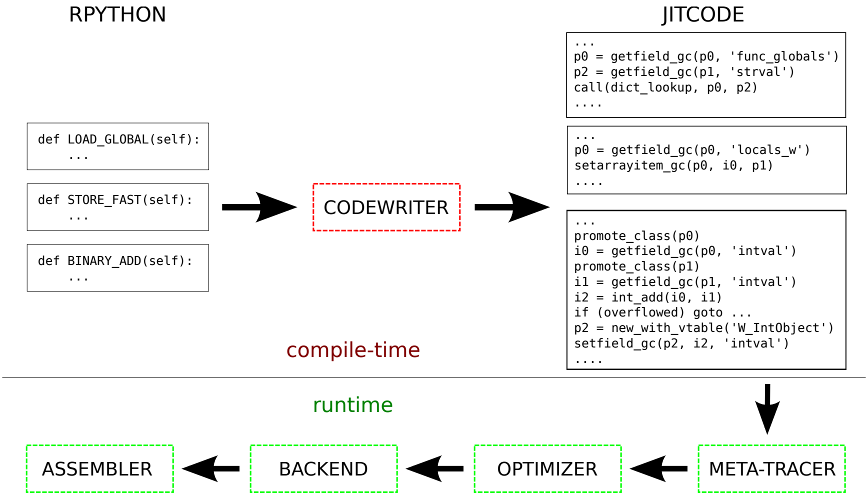
runtime

META-TRACER

OPTIMIZER

BACKEND

General architecture



PyPy trace example

```
def fn():
    c = a+b
    ...
```

PyPy trace example

```
def fn():
    c = a+b
    ...
```

```
LOAD_GLOBAL A
LOAD_GLOBAL B
BINARY_ADD
STORE_FAST  C
```

PyPy trace example

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
```

```
def fn():
    c = a+b
    ...
```

```
LOAD_GLOBAL A
LOAD_GLOBAL B
BINARY_ADD
STORE_FAST  C
```

PyPy trace example

```
def fn():
    c = a+b
    ...
```

```
LOAD_GLOBAL A
LOAD_GLOBAL B
BINARY_ADD
STORE_FAST  C
```

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
```

PyPy trace example

```
def fn():
    c = a+b
    ...

LOAD_GLOBAL A
LOAD_GLOBAL B
BINARY_ADD
STORE_FAST  C
```

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
```

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
```

```
...
guard_class(p0, W_IntObject)
i0 = getfield_gc(p0, 'intval')
guard_class(p1, W_IntObject)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(00, i1)
guard_not_overflow()
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
...
```

PyPy trace example

```
def fn():
    c = a+b
    ...

LOAD_GLOBAL A
LOAD_GLOBAL B
BINARY_ADD
STORE_FAST  C
```

```
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
...
p0 = getfield_gc(p0, 'func_globals')
p2 = getfield_gc(p1, 'strval')
call(dict_lookup, p0, p2)
...
...
guard_class(p0, W_IntObject)
i0 = getfield_gc(p0, 'intval')
guard_class(p1, W_IntObject)
i1 = getfield_gc(p1, 'intval')
i2 = int_add(00, i1)
guard_not_overflow()
p2 = new_with_vtable('W_IntObject')
setfield_gc(p2, i2, 'intval')
...
...
p0 = getfield_gc(p0, 'locals_w')
setarrayitem_gc(p0, i0, p1)
....
```

PyPy optimizer

- intbounds
- constant folding / pure operations
- virtuals
- string optimizations
- heap (multiple get/setfield, etc)
- unroll

Intbound optimization (1)

intbound.py

```python
def fn():
    i = 0
    while i < 5000:
        i += 2
    return i
```

Intbound optimization (2)

unoptimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add_ovf(i15, 2)
guard_no_overflow()
...
```

optimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add(i15, 2)
...
```

- It works **often**
- array bound checking
- intbound info propagates all over the trace

Intbound optimization (2)

unoptimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add_ovf(i15, 2)
guard_no_overflow()
...
```

optimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add(i15, 2)
...
```

- It works **often**
- array bound checking
- intbound info propagates all over the trace

Intbound optimization (2)

unoptimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add_ovf(i15, 2)
guard_no_overflow()
...
```

optimized

```
...
i17 = int_lt(i15, 5000)
guard_true(i17)
i19 = int_add(i15, 2)
...
```

- It works **often**
- array bound checking
- intbound info propagates all over the trace

Virtuals (1)

virtuals.py

```python
def fn():
    i = 0
    while i < 5000:
        i += 2
    return i
```

Virtuals (2)

unoptimized

```
...
guard_class(p0, W_IntObject)
i1 = getfield_pure(p0, 'intval')
i2 = int_add(i1, 2)
p3 = new(W_IntObject)
setfield_gc(p3, i2, 'intval')
...
```

optimized

```
...
i2 = int_add(i1, 2)
...
```

- The most important optimization (TM)
- It works both inside the trace and across the loop
- It works for tons of cases
  - e.g. function frames

Virtuals (2)

unoptimized

```
...
guard_class(p0, W_IntObject)
i1 = getfield_pure(p0, 'intval')
i2 = int_add(i1, 2)
p3 = new(W_IntObject)
setfield_gc(p3, i2, 'intval')
...
```

optimized

```
...
i2 = int_add(i1, 2)
...
```

- The most important optimization (TM)
- It works both inside the trace and across the loop
- It works for tons of cases
  - e.g. function frames

Virtuals (2)

unoptimized

```
...
guard_class(p0, W_IntObject)
i1 = getfield_pure(p0, 'intval')
i2 = int_add(i1, 2)
p3 = new(W_IntObject)
setfield_gc(p3, i2, 'intval')
...
```

optimized

```
...
i2 = int_add(i1, 2)
...
```

- The most important optimization (TM)
- It works both inside the trace and across the loop
- It works for tons of cases
  - e.g. function frames

Constant folding (1)

constfold.py

```python
def fn():
    i = 0
    while i < 5000:
        i += 2
    return i
```

Constant folding (2)

unoptimized

```
...
i1 = getfield_pure(p0, 'intval')
i2 = getfield_pure(<W_Int(2)>,
                   'intval')
i3 = int_add(i1, i2)
...
```

optimized

```
...
i1 = getfield_pure(p0, 'intval')
i3 = int_add(i1, 2)
...
```

- It "finishes the job"
- Works well together with other optimizations (e.g. virtuals)
- It also does "normal, boring, static" constant-folding

Constant folding (2)

unoptimized

```
...
i1 = getfield_pure(p0, 'intval')
i2 = getfield_pure(<W_Int(2)>,
                   'intval')
i3 = int_add(i1, i2)
...
```

optimized

```
...
i1 = getfield_pure(p0, 'intval')
i3 = int_add(i1, 2)
...
```

- It "finishes the job"
- Works well together with other optimizations (e.g. virtuals)
- It also does "normal, boring, static" constant-folding

Constant folding (2)

unoptimized

```
...
i1 = getfield_pure(p0, 'intval')
i2 = getfield_pure(<W_Int(2)>,
                   'intval')
i3 = int_add(i1, i2)
...
```

optimized

```
...
i1 = getfield_pure(p0, 'intval')
i3 = int_add(i1, 2)
...
```

- It "finishes the job"
- Works well together with other optimizations (e.g. virtuals)
- It also does "normal, boring, static" constant-folding

Out of line guards (1)

outoflineguards.py

```
N = 2
def fn():
    i = 0
    while i < 5000:
        i += N
    return i
```

Out of line guards (2)

unoptimized

```
...
quasiimmut_field(<Cell>, 'val')
guard_not_invalidated()
p0 = getfield_gc(<Cell>, 'val')
...
i2 = getfield_pure(p0, 'intval')
i3 = int_add(i1, i2)
```

optimized

```
...
guard_not_invalidated()
...
i3 = int_add(i1, 2)
...
```

- Python is too dynamic, but we don't care :-)
- No overhead in assembler code
- Used a bit "everywhere"

Out of line guards (2)

unoptimized

```
...
quasiimmut_field(<Cell>, 'val')
guard_not_invalidated()
p0 = getfield_gc(<Cell>, 'val')
...
i2 = getfield_pure(p0, 'intval')
i3 = int_add(i1, i2)
```

optimized

```
...
guard_not_invalidated()
...
i3 = int_add(i1, 2)
...
```

- Python is too dynamic, but we don't care :-)
- No overhead in assembler code
- Used a bit "everywhere"

Out of line guards (2)

unoptimized

```
...
quasiimmut_field(<Cell>, 'val')
guard_not_invalidated()
p0 = getfield_gc(<Cell>, 'val')
...
i2 = getfield_pure(p0, 'intval')
i3 = int_add(i1, i2)
```

optimized

```
...
guard_not_invalidated()
...
i3 = int_add(i1, 2)
...
```

- Python is too dynamic, but we don't care :-)
- No overhead in assembler code
- Used a bit "everywhere"

Guards

- guard_true
- guard_false
- guard_class
- guard_no_overflow
- **guard_value**

Promotion

- guard_value
- specialize code
- make sure not to **overspecialize**
- example: type of objects
- example: function code objects, ...

Conclusion

- PyPy is cool :-)
- Any question?