Dr Wenowdis: Specializing dynamic language C extensions using type information

SOAP 2024, June 25
Maxwell Bernstein
CF Bolz-Tereick

Image from Saturday Night Live

This talk

1. State of Python and C extensions
2. Oof, it’s slow in PyPy
3. Look, we made it faster in PyPy!
4. A rising tide lifts all ships

The state of things

Python’s continued success: glue C libraries together

Motivation: Python and C extensions

CPython was not focused on performance for the majority of its life

30+ years... many C extensions

An official C API now exists around `PyObject*`, types, functions

Think: Java Native Interface (JNI) designed to support multiple JVM implementations

An example C API function

```
long PyLong_AsLong(PyObject *obj)
```

Part of the Stable ABI.

... some more stuff ...

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

But there are other Pythons

PyPy: make Python fast

Advanced JIT compiler

Supports Python C extensions

pypy

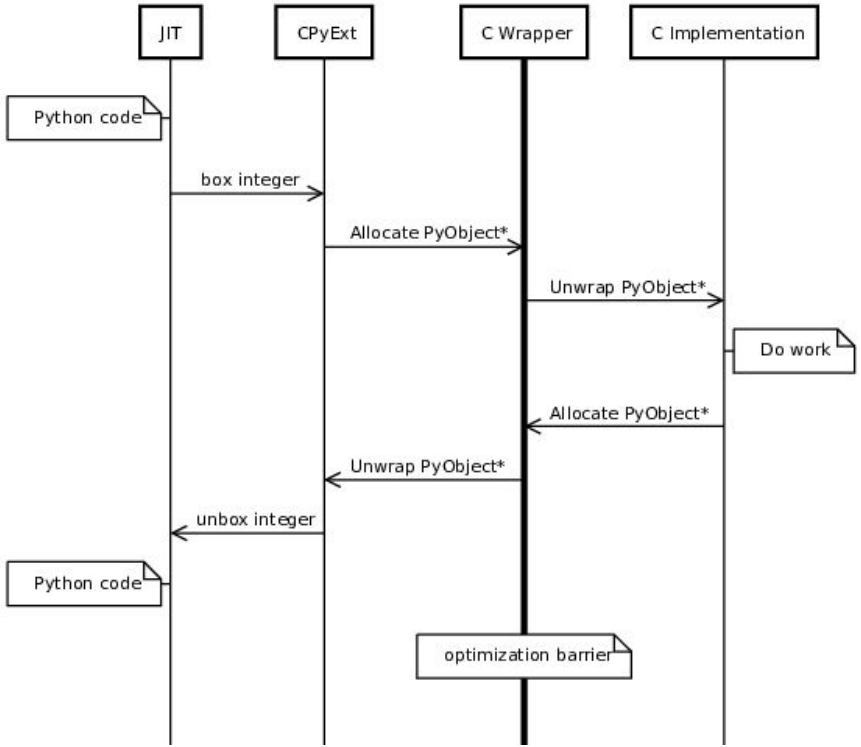
Motivation: why the C API hurts in PyPy

The CPython C API was designed to wrap

PyPy has a totally different object model:

- moving GC
- smaller objects with different layouts

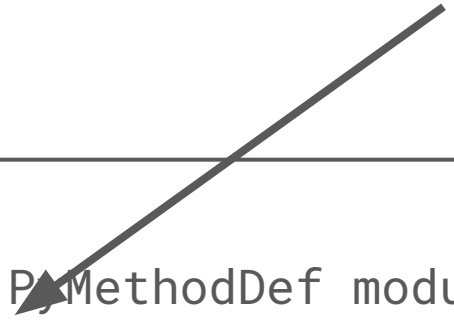
Overall **not built around** `PyObject*`



Let’s look at some Python code

```
import a_c_extension

print(a_c_extension.inc(3))  # => 4
```

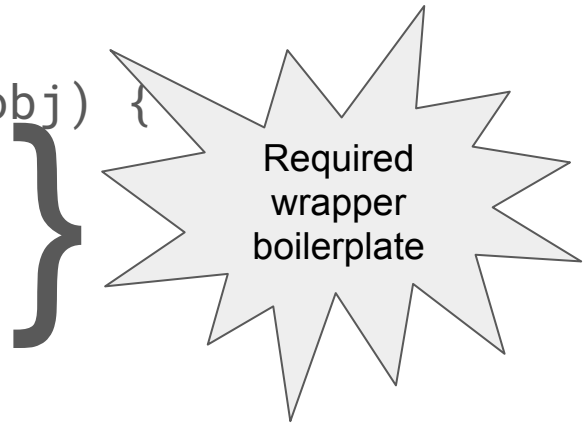


```
static PyMethodDef module_methods[] = {
    {“inc”, inc, METH_O, "Add one to an int"},
    {NULL, NULL, 0, NULL}
};
```

The C module has arg checking wrappers

```
long inc_impl(long arg) {
  return arg+1;
}

PyObject* inc(PyObject* module, PyObject* obj) {
  long obj_int = PyLong_AsLong(obj);
  if (obj_int == -1 && PyErr_Occurred()) {
    return NULL;
  }
  long result = inc_impl(obj_int);
  return PyLong_FromLong(result);
}
```



METH_FASTCALL, for more arguments

```
PyObject* add(PyObject* m, PyObject** args, Py_ssize_t nargs) {
  if (nargs != 2) return PyErr_Format(PyExc_TypeError, ...);
  if (!PyFloat_CheckExact(args[0])) { ... }
  double left = PyFloat_AsDouble(args[0]);
  if (PyErr_Occurred()) { ... }
  if (!PyFloat_CheckExact(args[1])) { ... }
  double right = PyFloat_AsDouble(args[1]);
  if (PyErr_Occurred()) { ... }
  double result = add_impl(left, right);
  return PyFloat_FromDouble(result);
}
```

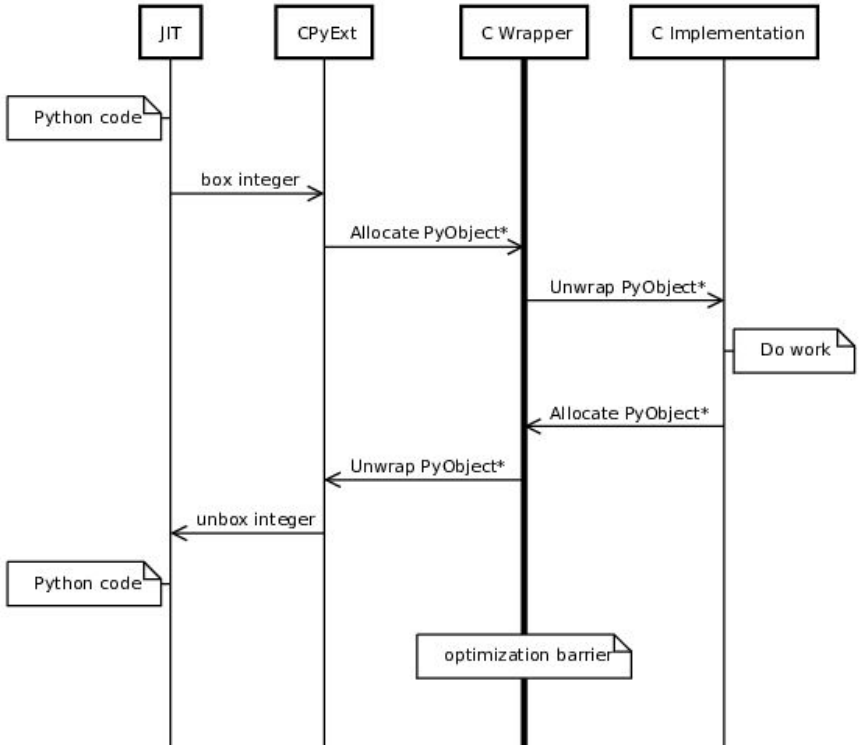
Problem: impedance mismatch

PyPy has probably already traced the code and its analyzer knows stuff about the function call

But we're checking all this stuff in pre-compiled C code

PyPy can't remove the checks!

Also, it still has to manufacture `PyObject*` since it cannot look inside the C code



```
int inc_arg_types[] = {T_C_LONG, -1};

PyPyTypedMethodMetadata inc_sig = {
  .arg_types = inc_arg_types,
  .ret_type = T_C_LONG,
  .underlying_func = inc_impl,
  .ml_name = "inc",
};

// c_long -> c_long with an exposed implementation

static PyMethodDef module_methods[] = {
   {inc_sig.ml_name, inc, METH_O|METH_TYPED, "..."},
   {NULL, NULL, 0, NULL}
};
```

How to fix: add types!

Where?? Stuffed *behind* the existing method metadata!

How to use the types in the JIT

Two step process:

1. In the optimizer, read C type signature
2. If argument types are known and match, call the underlying function instead

```
if (is_int(x)) {
  unboxed_result = (*fptr)(unbox(x))
  return box(unboxed_result)
}
```

How to use the types in the JIT

Two step process:

1. In the optimizer, read C type signature
2. If argument types are known and match, call the underlying function instead

```
if (is_int(x)) {
  unboxed_result = (*fptr)(unbox(x))
  return box(unboxed_result)
}
```

How to use the types in the JIT

Two step process:

1. In the optimizer, read C type signature
2. If argument types are known and match, call the underlying function instead

```
if (is_int(x)) {
  unboxed_result = (*fptr)(unbox(x))
  return box(unboxed_result)
}
```

How to use the types in the JIT

Two step process:

1. In the optimizer, read C type signature
2. If argument types are known and match, call the underlying function instead

```
if (is_int(x)) {
  unboxed_result = (*fptr)(unbox(x))
  return box(unboxed_result)
}
```

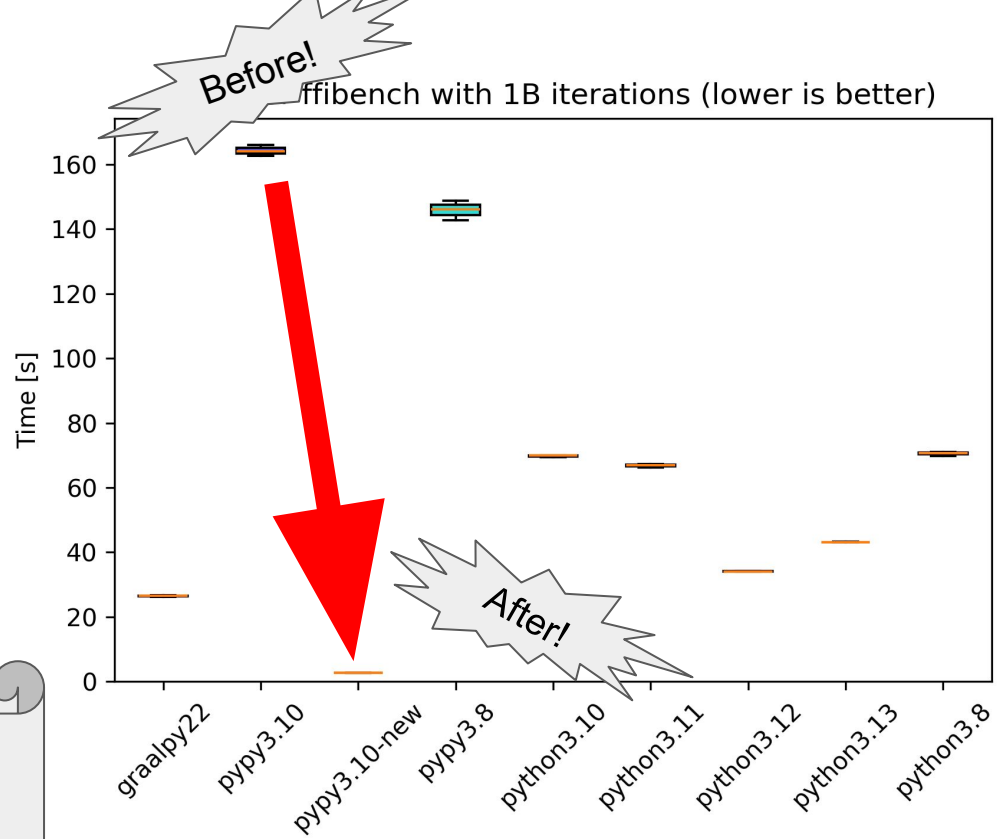
Some microbenchmark

```
import a_c_extension


def main(n):
    i = 0
    while i < n:
        i = a_c_extension.inc(i)
    return i

main(1_000_000_000)
```

That's a **big** win!



Consequences for PyPy

Correctness: opt-in

- Only kicks in if you have a bit set in your C extension
- Nobody has this bit set right now

Performance: opt-in

- C extensions that have been annotated get faster (remember: 60-80x*!)
- Nothing else changes

* in microbenchmarks

Consequences for other Python VMs

Already bites GraalPy, others

There is no impedance mismatch in CPython... yet

Once CPython JIT gets more advanced, they might have unboxed numbers

Takeaways

https://dl.acm.org/doi/10.1145/3652588.3663316

max@bernsteinbear.com

- Optimizing across language boundaries is hard
- Adding type information can get you pretty far
  - C function types? Wenowdis
- Surprisingly, doesn't break things
- Future: Emit from Cython / PyO3

```
cimport a_c_extension

def main(int n) -> int:
    i = 0
    while i < n:
        i = a_c_extension.inc(i)
    return i
```

Image from Saturday Night Live