f2pypy

There's a bit of discussion going on about the role of PyPy in scientific computing with Python. I spent a few days of the last week to add more ... shall I say "fuel?" .. to the discussion. I wrote a new back-end to f2py called f2pypy which generates a Python module to a shared library based on using ctypes. The module works (somewhat) with CPython, and does not work with PyPy because there's no way yet to pass a pointer to the array data to a ctypes function. (That's a minor detail which isn't hard to implement.)

What it shows is a real mechanism to get PyPy to support existing Fortran libraries already supported by f2py definition files.

NumPy isn't used in all scientific software

There is definitely place for PyPy in scientific computing even now. There are entire branches of science which have little overlap with the strengths of SciPy. I've been a full-time software developer for computational chemistry for 16 years, and have only used NumPy a few times.

One time I needed to compute the generalized inverse matrix. It was in a command-line program called by another process, of all things, and to my annoyance the "import numpy" on the cluster file system was noticably long. I forgot what the numbers were then, but the current numpy import adds 145 (yes, 145!) modules to sys.modules, and 107 of them start with "numpy." Our Lustre configuration did poorly with file metadata, and I think it was over a second to do the import.

I brought this up on the numpy list. While they made some changes, it was pointed out that I am not their target user. The "import numpy" also does an "import numpy.testing" and "input numpy.ctypeslib" and the other imports so that people could use numpy.submodule without an extra explicit import line, and because most people working with numpy are in working in a long-lived interactive session or job, so the startup performance isn't a problem.

I happen to disagree with their choice. "Explicit is better than implicit" and all that. But my point is not to argue for them to change but to give a specific example of how the goals of the NumPy developers can be different than the goals of other scientific programmers.

How do I use Python in science research?

A lot of what I do involves communicating with command-line executables. These are often written by scientists, and most are designed to be run directly by people, and not be other software. Most of the time is spent in the executable, so it doesn't matter if I'm using CPython or PyPy.

There are several cheminformatics libraries for Python. OpenBabel and OEChem use SWIG bindings, RDKit uses Boost, and I don't know what Indigo and Canvas use. Migrating these to PyPy will be hard. I hope that someone is working on SWIG bindings, but it looks like the PyPy developers don't want to commit to a C ABI. (See below.)

There's also code where there are no libraries, and for those I write the code in Python, and sometimes my own extension for C. For some of these case the 3x and higher performance of PyPy would be great. I also know a lot of ways for my CPython-based code to talk to PyPy-based code.

I used to develop software for bioinformatics and structural biology, and my observations still hold for those fields. One of the Biopython developers, Peter Cock, writes:

> Regarding Biopython using NumPy, we're already trying it out under PyPy. Large chunks of Biopython do not use NumPy at all, although there a few problems on PyPy 1.6 (one due to a missing XML library, bug filed), most of that seems to work." [*]

He continues with a list of some of what doesn't work.

Support for existing libraries

That said, I know that a lot of people depend Python bindings to existing libraries. These use the C API directly, or through auto-generated interfaces from f2py, Cython, Boost, SWIG, and many more. There's been 10+ years to develop these tools for CPython, and still very little time to adapt them to PyPy.

Relatively few extensions use the ctypes module, which is Python's "other" mechanism for calling external functions. Unlike the C API, this one is also portable across Jython, Iron Python, and PyPy. Obviously, if everyone used ctypes then there wouldn't be a problem. Why don't they?

One is the performance. Calling math.cos() is 8 times faster than doing a LoadLibrary() of libm and calling cos() that way. This is of course the worst case. But that's a CPython limitation. Pypy's ctypes call interface is faster than CPython calling a C extension:

```
% cat x.py
import ctypes
m = ctypes.cdll.LoadLibrary("/usr/lib/libm.dylib")
cos = m.cos
cos.argtypes = [ctypes.c_double]
cos.restype = ctypes.c_double

% python -mtimeit -s "from x import cos" "cos(0)"
1000000 loops, best of 3: 0.676 usec per loop
% python -mtimeit -s "from math import cos" "cos(0)"
10000000 loops, best of 3: 0.0811 usec per loop
% pypy -mtimeit -s "from x import cos" "cos(0)"
10000000 loops, best of 3: 0.0332 usec per loop
% pypy -mtimeit -s "from math import cos" "cos(0)"
```

```
100000000 loops, best of 3: 0.0047 usec per loop
```

although you can see it's still slower than using a built-in function.

Another reason to not use ctypes is that C/C++ library authors do interesting things with the API. One library I used has public API functions like "dt_charge(atom)" to get the formal charge of an atom, but used a number of #define statements to change those names to the internal name. That example became "dt_e_charge". It also defined certain constants only in the header files. This information isn't in the shared library.

I know at least one vendor which only ships a static library, and not a shared library. Apparently bad LD_LIBRARY_PATHs was such a support headache that they decided it wasn't worth it. (I think they are right.) There's no way to get ctypes to interface to a static library.

A fourth problem is lack of support for C++ templates. That clearly needs a compiler, which ctypes doesn't do.

PyPy needs a (semi-)stable C ABI; can you help?

Based on the above, there will clearly always be a need for compiler-based Python extensions, including PyPy extensions. That means there needs to be some sort of ABI that those extensions can program against.

I don't know what that would look like, and I think the PyPy developers think it's still too early to stablize on it. It may well be; but I think it's because there's no one on the group who wants to work on the task.

They were more than happy last year to show a proof-of-concept interface from PyPy to C++ using the run-time type information added by the Reflex system. (Yeah, I had never heard of it either.) So they have nothing against working with an existing ABI. Do you want to offer one?

I wrote "semi-" in the title because it wasn't until Python 3.2 that CPython got a stable ABI. PyPy notably does have emulation support for some of the CPython 2.x ABI but there are problems. Some modules use the ABI incorrectly, and it works for implementation-specific reasons. (For example, bad reference counts.)

If you are going to work on this, I think it would make sense to target the 3.2 ABI and to include instrumentation to help identify these problems.

The best for me would be if you develop some SWIG/ABI interface. This might just be to produce a bunch of stub functions and a ctypes definition for them. (Hmm, wasn't there a C++ to C SWIG interface?)

f2pypy: Experimental Fortran bindings

The above is talk and hand-waving. Code's also good. There was a PyPy sprint this week and I decided to join in for a few days and prototype an idea I've been thinking about: f2pypy. It's a variation of f2py which generates Python ctypes bindings which PyPy could use to talk with shared libraries implemented in Fortran.

At the end of several days of work, I got f2pypy to generate a Python module based on the "fblas.pyf" code from SciPy. I could import that library in CPython and (for the few functions I tested) get answers which matched the fblas module in SciPy. I could also use pypy to call *some* of the functions, but PyPy's "numpy" implementation is not mature enough. Its array objects don't yet support the ctypes interface, so I was unable to call out to the shared library. I could only call the scalar-based functions.

The code is definitely incomplete. Even my CPython-based tests fail some of the the "test_blas.py" from SciPy (I don't implement "cblas" and I think one of the tests depends on Fortran order instead of C order.) It's a proof-of-concept which shows that this approach is definitely viable, and it shows some of the difficulties in the approach.

My point though is that it opens new possibilites which aren't available in NumPy. For example, suppose you want to use one of the BLAS functions in your code. Every Mac includes a copy of BLAS as a built-in library. Instead of making people install SciPy, what about shipping the ctypes module description instead, and using that interface? You can ship pure Python code and still take advantage of platform-optimized libraries!

I earlier highlighted the performance problems in CPython's ctypes interface. But this is PyPy. They already have cross-module optimizations for Python calling Python. There's no reason why those can't apply to ctypes-based functions. (Or perhaps it's already there? I've not tested that.)

How does it work?

Fortran bindings are nice because they don't have the same preprocessor tricks that I mentioned earlier. Pearu Peterson wrote the excellent f2py package starting some 10+ years ago. It has several ways to work with Fortran code. The one I used was to start with a "pyf" definition file and generate Python code using a new back-end.

I figured out how to get SciPy to generate the pyf file for the BLAS library. (The SciPy source uses a template language during the build process to generate the actual code.) I used f2py's "crackfortran" module to parse the pfy file and get the AST. It's a small tree so perhaps I should call it an abstract syntax bush.

The f2py code generate the Python/C extension code based on the AST. My f2pypy code is basically another back-end, which generates ctypes-based code in Python.

The trickiest part was support for C code. Some of the pyf definition lines contain embedded C code. Here I've gathered three examples:

```
integer optional, intent(in),check(incx>0||incx<0) :: incx = 1
integer optional,intent(in),depend(x,incx,offx,y,incy,offy) :: n = (len(x)-offx)/abs(incx)
callstatement (*f2py_func)((trans?(trans==2?"C":"T"):"N"),&m;,&n;,α,a,&m;,x+offx,&incx;,β,y+offy,&incy;)
```

I used Fredrick Lundh's wonderful essay on Simple Top-Down Parsing in Python to build a simple C expression parser, which builds another AST. With a bit of AST manipulation, and symbol table knowledge (I need to know which inputs are scalars and which are vectors), I could generate output strings like:

```
def srot(..., incx = None, ...):
  ...
  if incx is None:
    incx = _ct.c_int(1)
  else:
    incx = _ct.c_int(incx)
  if not ((((incx.value) > 0) or ((incx.value) < 0))):
    raise ValueError('(incx>0||incx<0) failed for argument incx: incx=%s' % incx.value)
```

and the more complicated:

```
_api_cgemv((("c") if (((trans.value) == 2)) else ("t")) if ((trans.value)) else
("n"), (m), (n), (alpha), a.ctypes.data_as(_ct.POINTER(_complex_float)), (m),
(x if ((offx.value)) == 0 else x[(offx.value):]).ctypes.data_as(_ct.POINTER(_complex_float)),
(incx), (beta),
(y if ((offy.value)) == 0 else y[(offy.value):]).ctypes.data_as(_ct.POINTER(_complex_float)),
(incy))
```

I definitely do not generate optimized code. I decided to work completely in terms of ctypes scalars and numpy arrays, even for the check() statements. PyPy doesn't optimize that yet, and I think someone else could do a better job by only doing the conversion as part of the call to the Fortran code.

Usage

To generate the new module on a Mac (I don't know the shared library name for other OS installations):

```
  $PYTHON -m f2pypy tests/fblas.pyf -l vecLib --skip cdotu,zdotu,cdotc,zdotc
```

This generates "fblas.py". I have some test code for that module

```
% python test_fblas.py
python test_fblas.py
..........F...
======================================================================
FAIL: test_srot_overwrite (__main__.CBlasTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_fblas.py", line 116, in test_srot_overwrite
    assert x is x2
AssertionError

----------------------------------------------------------------------
Ran 15 tests in 0.006s

FAILED (failures=1)
```

This says that "numpy.array(.. copy=False)" makes a new reference, while the internal code f2py uses passes back the same object, so a real implementation will need to handle that detail.

Here's the same output from pypy:

```
% pypy test_fblas.py
.EE.EEEFEEEE.E.
======================================================================
ERROR: test_dnrm2 (__main__.CBlasTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_fblas.py", line 166, in test_dnrm2
    E(m.dnrm2(x), float((numpy.array([1+1+16+81], "d")**0.5)))
  File "/Users/dalke/cvses/f2pypy/fblas.py", line 1192, in dnrm2
    return _api_dnrm2((n), (x if ((offx.value)) == 0 else x[(offx.value):]).ctypes.data_as(_ct.POINTER(_ct.c_c
AttributeError: 'numarray' object has no attribute 'ctypes'

======================================================================
ERROR: test_drot (__main__.CBlasTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_fblas.py", line 153, in test_drot
    E(m.drot(1,2,3,4), (numpy.array(11.0, dtype="d"), numpy.array(2.0, dtype="d")))
  File "/Users/dalke/cvses/f2pypy/fblas.py", line 181, in drot
    y = _np.array(y, 'd', copy=overwrite_y)
TypeError: __new__() got an unexpected keyword argument 'copy'
 ....
```

See the dots up there with the "E"rrors? That 4 of the scalar-based tests pass. What fails is the vector-based code. The "ctypes" gets the pointer to the numpy array data, and PyPy doesn't support the "copy" parameter of numpy.array.

Still, it does pass some tests!

Future

I don't use Fortran modules. I don't use f2py. I don't use numarray. I will not be involved in this project for the future. (I do a lot of integration work, and I do a lot of parsing and AST transformations, so that part of this effort was a very pretty good fit!)

I did this because I wanted to show that PyPy can support traditional numeric software libraries and that there is a relatively doable path for migration from existing numpy code to "numpypy" code.

I will not be maintaining the project in the future. If you want to take it on, feel free. I've contributed it to the PyPy project, and it has its own repository. Feel free also to leave a comment or ask me questions.

Andrew Dalke is an independent consultant focusing on software development for computational chemistry and biology. Need contract programming, help, or training? Contact me

---