Optimizing Python with Pyrex

Pyrex is cool but not magic

- Pyrex very seldom makes pure-Python code faster just through a recompile

- But if you understand how Pyrex works, you can dramatically improve Python program performance:
  - use static type checking,
  - static binding,
  - C calling conventions, etc.

Case Study 1: Bisect Module

Bisect module

- Originally coded in Python.
- Recoded in C recently for performance.
- Pure Python version is between 2 and 3 times slower than C version.
- Recompile as Pyrex improves performance by only a few percentage points.

http://www.prescod.net/python/pyrexopt/bisect/rxbisect0.pyx

Representative function

```
def bisect_left(a, x, lo=0, hi=None):
  if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)/2
        if a[mid] < x: lo = mid+1
        else: hi = mid
    return lo
```

Optimization step 1: Static types

```
cdef int internal_bisect_left(a, x,
  int lo, int hi) except -1:
     cdef int mid
     while lo < hi:
          mid = (lo+hi)/2
          if a[mid] < x: lo = mid + 1
          else: hi = mid
     return lo
```

http://www.prescod.net/python/pyrexopt/bisect/rxbisect1.pyx

Why are we still slower?

- Pyrex calls PyObject_GetItem/SetItem rather than PySequence_GetItem/SetItem
- You can see the difference just in the type signatures!

```
PyObject_GetItem(PyObject *o, PyObject *key);

PySequence_GetItem(PyObject *o, int i);
```

Optimization step 2: Cheat

- Import and use PySequence_GetItem directly from Python.h

```
if PySequence_GetItem(a,mid) < x:
  lo = mid + 1
else:
  hi = mid
```

- http://www.prescod.net/python/pyrexopt/bisect/rxbisect2.pyx

Can we automate this?

- I propose to change Pyrex to have first-class support for “sequence” types.
- I wrote a patch that does this.

Pyrex with sequence types

- Now my code looks like this:

```
cdef int
  internal_bisect_left(sequence a, x,
  int lo, int hi) except -1:
  cdef int mid
     while lo < hi:
          mid = (lo+hi)/2
          if a[mid] < x: lo = mid + 1
          else: hi = mid
     return lo
```

- http://www.prescod.net/python/pyrexopt/bisect/rxbisect3.pyx

One more twist

- The C bisect module has this code:

```
if (PyList_Check(list)) {
    if (PyList_Insert(list, index,
  item) < 0)
        return NULL;
} else {
    if (PyObject_CallMethod(list,
  "insert", "iO", index, item) ==
  NULL)
        return NULL;
}
```

Pyrex can do that too!

```
if PyList_Check(a):
   PyList_Insert(a, lo, x)
else:
   a.insert(lo, x)
```

- By the way, note how two lines of Pyrex equal approximately 6 of C.
- And yet they do all of the same error and reference checking!

Result

- Even so, Pyrex seems to come out ~25% slower than C. :(
- But half as many lines of code!
- No decrefs!
- No goto statements!
- Automatic exception propagation!

- Seems like a good trade-off to me!

Case Study 2: Fibonnaci

A digression on benchmarking

Be careful

- As always, you have to be careful benchmarking and optimizing.
- For instance: GCC compiler flags can make all of the difference.
- Furthermore, applying them is not always straightforward.

Fibonacci optimizations

Opt Level	Pyrex	Plain C
None	0m25.790s	0m18.180s
-O	0m13.990s	0m13.370s
-O2	0m15.450s	0m9.440s
-O3	0m9.720s	0m5.840s
-O3 -fun…*	0m7.430s	0m4.730s

(* -fun… = -funroll-loops)

Case Study 3: Heap Queue

Heap queue

- Similar story to Bisect.
- But even with sequence types added, Pyrex loses out.
- heapq.c uses some highly optimized PyList APIs:

```
#define PyList_GET_ITEM(op, i)
  (((PyListObject *)(op))->ob_item[i])
```

Pyrex can cheat too!

```
cdef extern from "Python.h":
    int PyList_Check(object PyObj)
    int PyList_GET_SIZE(object PyList)

    cdef void *PyList_GET_ITEM(object PyList, int
  idx)
…

lastelt = <object>PyList_GET_ITEM(lst, length-1)
```

http://www.prescod.net/python/pyrexopt/heapq/rxheapq4.pyx

Why does this work?

- Pyrex compiles to C.
- The C preprocessor will recognize calls to `PyList_GET_ITEM`
- Note that you have to be careful about refcounts!

Results

- With every cheat I could think of…
  - Crushed Python (as much as 6 times faster depending on the operation)
  - Didn't quite rival C (30-60% slower).
  - Maybe somebody smarter than me can find some optimizations I missed…
- But no code like this:

```
tmp = PyList_GET_ITEM(heap, childpos);
Py_INCREF(tmp);
Py_DECREF(PyList_GET_ITEM(heap, pos));
PyList_SET_ITEM(heap, pos, tmp);
```

Case Study 4: Py(e)xpat

Real-world example

- Years ago I was one of several people to help wrap Expat in C.
- It wasn't rocket science but it was a pain.
- There were many niggly details.
  - E.g. a couple of 40 line functions to allow stack traces to propagate from callbacks through Python???
  - Ugly macros everywhere
  - Careful reference counting
  - Plus all of the usual C extension ugliness
- I tried again with Pyrex, yielding pyxpat.

Results

- The Pyrex version is consistently faster than the C version, but it may cheat a bit:
  - the C version implements some optional features that the Pyrex version does not.
  - in C, the benchmark doesn't *use* the features
  - in Pyrex, it doesn't even *have* the feature.
  - that means a few less conditionals and branches.
  - either way, Pyrex is impressively fast!

Statistics

- I parsed a file with the following statistics:
  - 400 MB of data
  - 5 million elements
  - 10 million startElement/endElement callbacks into pure Python code
  - it took 3 minutes on my 700MHZ PowerBook.
  - this exercise is *not* I/O bound or Pyrex couldn’t consistently beat the C API.

One step further

- Pyxpat can explicitly expose a callback interface that uses Pyrex/C calling conventions rather than Python?
- Without changing our logic or interface, we can slash the time in half!
- It turns out that half of Pyexpat's time is spent on the Python function call dance.
- And that's even ignoring type coercion issues.

Conclusion

Where does Pyrex fit in?

- Pyrex code isn't quite as fast as hand-coded C.
- Bu you can focus on algorithm rather than pointers and buffers. (sound familiar?)
- Pyrex code can live on a spectrum
  - from "almost as abstract as Python"
  - to "almost as efficient as C."
- The next best thing to the best of both worlds.

What are the obstacles?

- You will only get performance out of Pyrex if you understand implementation details.
- Pyrex should allow programmers to declare information relevant to the optimizer.
- Pyrex ought to generate more efficient code by default.

Optimizing Pyrex itself

- Pyrex itself is ripe for optimization:
  - More knowledge about Python container types
  - Better string interning
  - Static binding of globals and builtins (yes, Pyrex looks up len() in __builtins__)
  - Bypass `PyArg_ParseTuple` when there are no arguments
  - PyObject_NEW instead of PyObject)_CallObject

My thoughts

- Given how easy Pyrex is, I predict that Python+Pyrex programs will typically go faster than Python+C programs *given the same amount of programmer effort*.

- If you are building a system that needs some performance…try Pyrex.
- It is probably fast enough for anything short of a database or network stack.