Localized Type Inferencing in Python

Brett Cannon

Aaron Keen

California Polytechnic State University, San Luis Obispo

PyCon 2005

Type Inferencing

- *Tightest mapping of possible types to a variable*
- *Determined statically*
- *Not allowed to make wrong inference*
    - *compilation decisions based on this info*

Can type inferencing be added to Python's compiler for a performance increase?

No semantic changes to language or compiler allowed. Speed-up achieved from type-specific opcodes.

Hindley-Milner

- *Used in Standard ML and Haskell*
- *bottom-up or top-down algorithm*
- *Allows abstract types*
- *Cannot handle function arguments of other functions used in a polymorphic fashion*

Cartesian Product / Iterative Type Analysis

- *What Starkiller uses*
- *Iteratively try to find fixed point where types don't change*
- *Works with concrete types only*

The Compiler

- *Input of parse tree, output of bytecode*
    - *bytecode typeless sans list/dict/tuple creation*
- *Can be considered a self-contained program*
    - *i.e., does not use anything to base compilation on except parse tree*

The Problem

- *Does not check ‘import’ dependencies*
  - *Can compile code that imports non-existent modules*
  - *Can swap in different module than what was present at compile-time*
- *You can’t depend on what is contained in other modules*

The Language

- *Highly dynamic*
- *Injection into another module's global namespace allowed*
- *Tons of other ways to play with a variable's value at run-time*
    - *Standard library (tracing, frames, etc.) exacerbates situation*

The Other Problem

- *An external module can inject/replace objects in a module's global namespace*

What This All Means

- *Since another module can change a module's global namespace and we can't know anything about another module at compile-time*
- *Everything at the global level must be considered unknown*

Can’t infer squat!

Or can we ?

Atomic Types in Local Scope

- *Any type that is syntactically supported and defined locally*
  - *integrals (int, long)*
  - *floats*
  - *complex numbers*
  - *basestring (str, Unicode)*
  - *lists*
  - *tuples*
  - *dicts*

The Algorithm

Implemented using Python 2.3.4

‘if’ Statement

```
a = 1        # a = (integral,)
if foo:      # a = (integral,)
  a = [a, 2]     # a = (list,)
elif bar:    # a = (integral,)
  a = (a, 2)     # a = (tuple,)
elif baz:    # a = (integral,)
  pass
else:        # a (integral, )
  a = {0:a, 1:2} # a = (dict,)

# a = (integral, list, tuple, dict)
a[1]
```

Loops

```
a = 1         # a = (integral,)
for x in range(10):
  a + 3
  a = 1.0       # a = (float, integral)
else:         # a = (float, integral)
  a = 4+0j      # a = (complex,)

# a = (complex, float, integral) !
a / 2
```

try/except/finally/else

```
a = ()              # a = (tuple,)
try:                # a = (tuple,)
  a[0]
  a = []              # a = (tuple, list) !
except Exception:   # a = (tuple, list)
  pass
except:             # a = (tuple, list)
  a = {}              # a = (dict,)
else:               # a = (tuple, list) !
  a = “PyCon”         # a = (basestring,)

# a = (tuple, list, dict, basestring)
a[0]
```

Type Annotations

- *For functions or methods*
- *Stored in first line of comment for a function; “””::128::”””*
- *Done by hand*
- *Completely optional*
    - *Done to see if optional static type checking could give performance boost*

Other Tidbits

- *Closures properly supported*
- *Contents of tuples left unknown*
    - *simplified implementation*
- *Highest accuracy for ‘try’ block not done*
    - *for simplicity reasons*
- *Detect ‘break’?*

Choosing New Opcodes

- *Based on what types compiler could infer for various opcodes*
- *Used BitTorrent, Mailman, PIL, Plone, Pyrex, PythonCard, SciPy, Twisted, and the Python Standard Library*
- *Ranked based on:*
  - *raw count*
  - *count/LOC*

New Opcodes

Name	Replaces	Speedup
DICT_STORE	STORE_SUBSCR(dict, *, *)	3%
STR_FORMAT	BINARY_MODULE(basestring)	8%
LIST_APPEND	list.append()	39%
STR_CONCAT	BINARY_ADD(basestring, basestring)	8%
STR_MULT	BINARY_MULTIPLY(integral, basestring)	9%
STR_JOIN	basestring.join()	20%
INT_LSHIFT	BINARY_LSHIFT(integral, integral)	16%
DICT_GETITEM	BINARY_SUBSCR(dict, *)	6%
LIST_CMP	COMPARE_OP(*, list, list)	9%
DICT_HAS_KEY	dict.has_key()	51%

Benchmarks

- *SpamBayes*
- *Pyrex (with/without annotations)*
- *PyBench*
- *Parrotbench (with/without annotations)*

Results

SpamBayes	- 2.1%
PyBench	-0.2% (0.5%)
Pyrex (base)	1.0%
Pyrex (annotations)	1.6%
Parrotbench (base)	0.7%
Parrotbench (annotations)	0.8%

Also found 3 unit tests in Python Standard Library that were testing for things at run-time now caught at compile-time

but, overall ...

It ain’t worth it!

But if we changed some things ...

What Changes Could Help

- *“Unsimplify” implementation*
- *Timestamp/checksum import dependencies*
- *Specify when injecting over built-ins*

Questions?