

# *Starkiller: A Static Type Inferencer and Compiler for Python*

Michael Salib  
[msalib@alum.mit.edu](mailto:msalib@alum.mit.edu)

Dynamic Languages Group  
Computer Science & Artificial Intelligence Lab  
Massachusetts Institute of Technology

May 11, 2004

# *This talk in 60 seconds*

I. Motivation

II. Why Python is slow

III. Starkiller type inference

IV. Starkiller compilation

V. Results and Challenges

VI. Questions

# *I. Motivation*

Reason 38 for destroying the sun:

The sun reduces our dependance on foreign oil. It is unpatriotic.

# *The end of the world*

- ◆ Software sucks. A lot.
  - ◆ too buggy, too dangerous
  - ◆ too expensive and slow to build
  - ◆ too pervasive
  - ◆ internet makes it all worse
  - ◆ bad software kills people
  - ◆ it is going to get worse before it gets better

# *Saving the world: Python*

- ◆ Use a High level language
  - ◆ fewer lines of code needed
  - ◆ fewer lines mean
    - ◆ fewer bugs
    - ◆ less time/money to build
  - ◆ make the worst brain damage impossible
    - ◆ no buffer overflows in Python programs
- ◆ But Python cannot take over the world
  - ◆ no continuations
  - ◆ no macro system out of the box
  - ◆ too slow

# *Python is slow*

- ◆ I've done everything with Python
  - ◆ High speed network servers
  - ◆ Databases
  - ◆ Statistical natural language processing
  - ◆ Scientific computing
  - ◆ Signal and Image processing
  - ◆ AI type job schedulers
- ◆ And its been slow

# *Python is not slow!*

- ◆ You're a heretic!
- ◆ Most apps spend all their time waiting
  - ◆ on a socket (network servers)
  - ◆ on a slow human (GUIs)
  - ◆ on Oracle (databases)
  - ◆ on disk IO (most things)
- ◆ Fast libraries written in C/C++
- ◆ Numeric!
- ◆ Die infidel, die!

# *Yes, Python is slow*

- ◆ I've used all those lines myself
- ◆ I even believe them
- ◆ They're relevant most of the time
- ◆ But they don't change the fact that Python is slow
- ◆ Sometimes, straightforward Python code is much clearer and easier to write than fighting with Numeric
- ◆ For the 15% of apps where speed matters, pure Python can't do the job alone
- ◆ I don't want to use crappy C/C++



## *II. Why Python is slow*

Reason 347 for destroying the sun:

It warms our enemies.

# *Those who do not learn from history...*

- ◆ p2c was a python to C compiler emerged circa 1998
- ◆ It generated (lots of) C code that made the same calls into the Python runtime that the VM would
- ◆ But it compiled down to machine code!
- ◆ So it must be super fast!
- ◆ Super = 10-15%
- ◆ A lesson: the VM is not a performance bottleneck (yet)

# *Where should I jump now?*

- ◆ Quick! Inline the function `f` in the code below!
- ◆ A lesson: dynamic binding seemed like such a good idea at the time...

```
if random() > 0.5:  
    def f(x): return x + 1  
else:  
    def f(x): return x - 1  
print map(f, range(4096))
```

## *Trapped in a box*

- ◆ Numbers are heap allocated objects referenced by pointer; they are neither special nor unique snowflakes
- ◆ New coercion rules make life even worse: integer overflow silently coerces to longs
- ◆ A lesson: boxing replaces fast register ALU ops with multiple dereferences of distant (read: not in cache) memory

# *Our old (performance killing) friend...*

- ◆ Dynamic dispatch has a long history of ruining performance in OOP languages
- ◆ cf virtual/nonvirtual methods in C++, sealing in Dylan
- ◆ By postponing until runtime decisions about which bit of code is executed at a polymorphic call site, we lose the ability to optimize well
- ◆ You cannot inline code when you don't know what it is

# *More Pythonic “fun”*

- ◆ Multiple inheritance
- ◆ First class functions with lexical scoping
- ◆ No declarations or manifest types
- ◆ `getattr` and `setattr` functions allow anyone to get/set any attribute at runtime
- ◆ Dynamic inheritance relations
- ◆ Dynamic class membership
  - `x = table()`
  - `x.__class__ = chair`
  - `assert isinstance(x, chair)`

# *Other languages suck*

- ◆ Java sucks beyond all measure and comprehension
- ◆ C++ and Java suffer the same performance problems as Python when it comes to dynamic dispatch
- ◆ Dynamic dispatch prevents the compiler from using all the cool optimizations like inlining
- ◆ Inlining is the canary in the coal mine: if you can't inline, you probably can't do loop hoisting, strength reduction, etc.

### ***III. Starkiller type inference***

Reason 7 for destroying the sun:

The sun causes global warming.



# *Making Python fast*

- ◆ Speed == laziness: stop doing work
- ◆ Work refers to all the runtime choice points the Python VM has to perform
  - ◆ whenever the VM has to find what code to execute next
  - ◆ whenever the VM has to check operands to ensure they are of the correct type
- ◆ We can eliminate many of those checks using static analysis, specifically type inference

# *Finding the right pigeon hole*

- ◆ Compiling to C++ is not enough (cf p2c)
- ◆ Need static type inference to eliminate dynamic binding and dispatch
- ◆ Starkiller compliments rather than replaces CPython
- ◆ Covers the entire language except eval, exec, and dynamic module loading
- ◆ Not all run time choice points can be eliminated, but many can

# *Starkiller type inference*

- ◆ Based on Ole Agesen's Cartesian Product Algorithm (see his Stanford thesis)
- ◆ Represent Python programs as dataflow networks
- ◆ Nodes correspond to expressions and have a set of concrete types those expressions can achieve at runtime
- ◆ Constraints connect nodes together and enforce a subset relation between them
- ◆ Types flow along constraints

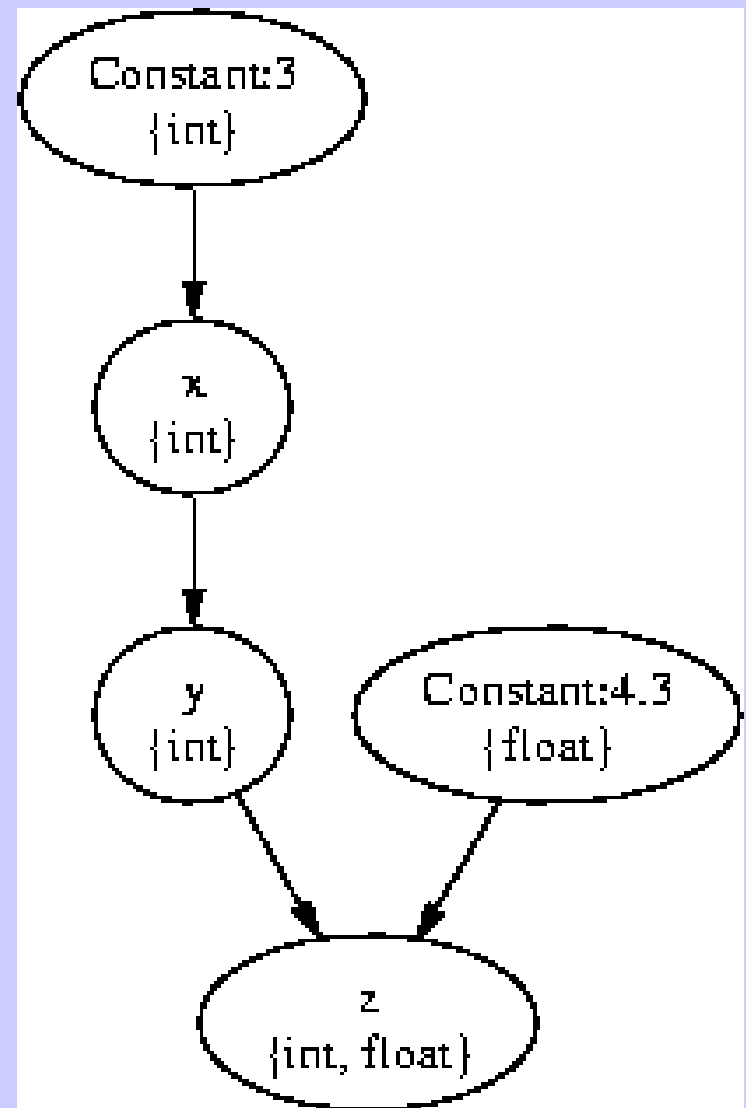
# *Ex-girlfriends say I'm insensitive*

- ◆ Starkiller's type inference algorithm is flow-insensitive
- ◆ It has no notion of time
- ◆ Code like `x = 3; doSomething(x); x = 4.3; doSomething(x)` will suffer loss of precision
- ◆ I don't care. I'm insensitive, remember?

# Type inference in action

## ◆ A simple example

x = 3  
y = x  
z = y  
z = 4.3



# *Functions and Templates*

- ◆ Parametric polymorphism (same function with different argument types) reduces precision
- ◆ We regain precision by taking cartesian product of argument type list and analyzing one template for each monomorphic argument list
- ◆ Given polymorphic calls `max(1, 2)` and `max(3.3, 4.9)`, we analyze templates for `(int, int)`, `(float, int)`, `(int, float)`, and `(float, float)`

# *Functions and Definitions*

- ◆ A Python function definition creates a first class object at runtime
- ◆ Function objects can capture variables defined in their lexical parent(s)
- ◆ Starkiller models function definition using a function definition node that has constraints from all default args and expressions the function closes over
- ◆ The definition node takes the cartesian product and generates monomorphic function types

# *Objects and Classes*

- ◆ Class definition works just like function definition!
- ◆ Instances work in the same way as classes!
- ◆ Calling a class triggers the creation of an instance definition node
- ◆ ID nodes are the repository for the polymorphic state of an instance
- ◆ They generate monomorphic instance state types and send them into the world



# *Foreign Code*

- ◆ Type inference cannot see into an extension module
- ◆ We could perform type inference on C/C++/Fortran...therein lies doom
- ◆ Starkiller gives extension writers a minilanguage for declaring the type inference properties of their extensions
- ◆ Most extensions are real simple: `int(x)` always returns an integer

# *Foreigner code, living among us, plotting against us!*

- ◆ Some extensions are unspeakably complicated
  - ◆ they might call arbitrary functions
  - ◆ they might mutate their arguments or some object that is part of global state
- ◆ The external type description language is really Python
  - ◆ External type descriptions run as extensions of the Starkiller type inferencer
  - ◆ You can use them to raise the dead

## *IV. Starkiller compilation*

Reason 204 for destroying the sun:

DARPA say sun bad. Must kill or lose funding.

# *Compilation preliminaries*

- ◆ Functions/classes/modules are represented by C++ objects that can be passed around
- ◆ Each function/method template gets compiled as a separate monomorphic block of code
- ◆ Since modules are executed exactly once, their attributes are all static
- ◆ Conservative GC thanks to Boehm
- ◆ No relation between Python and C++ object models

# *Data model*

- ◆ Numbers are automatically unboxed
- ◆ Everything else is heap allocated and passed by reference
- ◆ Container datatypes are built out of STL components and are type specific

# Closures

- ◆ Normally, variables are stack allocated
- ◆ But, for variables referenced by inner functions, Starkiller allocates them specially from a heap allocated MiniStackFrame
- ◆ An MST is common space that the original function and all of its inner functions can safely refer to, even after the original function returns
- ◆ The MST persists as long as it remains referenced thanks to the magic of GC

# *Fast Polymorphic dispatch*

- ◆ We cannot eliminate all of it
- ◆ Usually implemented with an indirect branch through a class pointer
  - ◆ very, very slow on modern hardware
- ◆ For the common case where there are few possibilities, we exploit the lack of eval to speed things up
- ◆ Use gcc's computed-goto extension plus minimal hashing to jump directly into the code without a branch

# *Dynamic attributes*

- ◆ `getattr` is easy to optimize:
  - ◆ use perfect hashing (plus extra if `setattr`)
- ◆ `setattr` contaminates objects
  - ◆ any attribute can be of the type assigned in the `setattr` call



# *Exceptions*

- ◆ All Starkiller defined functions/methods can throw `InternalRuntimeError`
- ◆ A Python `try/except` block is translated into a C++ `try/catch` block that dispatches on the exception thrown
- ◆ Python library code based on C++ components translate native C++ exceptions into their Python equivalent

# *Generators*

- ◆ Generators become functions that return instances of a generator object
- ◆ Variables in the generator body become attributes of the object
- ◆ yield statements get replaced by code that saves the label corresponding to the next statement to be executed and then returns
- ◆ On each invocation of the generator object, control jumps to the label last saved

## *V. Results and Challenges*

Reason 172 for destroying the sun:

The pale yellow face mocks us, keeps us from hearing the machine; it burns, it burns, we hatesss it!

# *Where are we now?*

- ◆ Starkiller type inferencer is mostly implemented
  - ◆ almost all of the hard parts are done
  - ◆ most of the unfinished work is boring detail
- ◆ The compiler is in the very early stages
  - ◆ a prototype works on simple code that doesn't push it too hard
  - ◆ no runtime system, no builtin types except int and float

# *Suckling on the government teat*

- ◆ Who owns Starkiller? MIT!
- ◆ Who paid for Starkiller's development?
- ◆ You did! Pat yourselves on the back!
- ◆ Thank you taxpayers
- ◆ “So, that means that you are a whore, MIT is your pimp, and DARPA is the john who likes to play rough. . .Hey Mike, is there anything you won't do for money?”
- ◆ A secret: don't tell DARPA I'm not building the sun destroying weapon they think I am

# *Justify your existence*

- ◆ Very preliminary benchmark with the prototype compiler and type inferencer
- ◆ All benchmarks are lies
- ◆ This one is pathological
- ◆ Call the factorial and fibonacci functions
- ◆ In a loop. Over and Over.
- ◆ CPython completion time: 18:37
- ◆ Starkiller completion time: 0:15
- ◆ Speedup: 60

# *Challenges*

- ◆ Static error detection
- ◆ Template shadowing
- ◆ False numeric polymorphism
- ◆ Partial evaluation
- ◆ Overflow coercion
- ◆ Restoring eval functionality

## *VI. Questions?*

- ◆ Because it seemed like a good idea at the time...