

Starkiller

MIT Artificial Intelligence Laboratory
Dynamic Languages Group

Mike Salib
msalib@mit.edu

August 10, 2001

Overview

- I. Python, Type Inference, and Why its h
- II. Starkiller: Goals, Design, and New ide
- III. Challenges and future work

I. Python . . .

- statically scoped
- dynamically typed
- no declarations
- lots of OOP
- every object has a type
- names are bound to objects in a namespace (dictionary)

More snake time

- Code is compiled to byte codes for a stack VM
- Core functionality in C extensions:
 - this is not a self hosting language
- Automatic MM: refcounting + occasional GC
- Object compatibility based on signature, not inheritance
- We're all consenting adults

Type inference

For static TI

- Use a constraint network
- Assign types for all constant entities
- Add constraints between nodes to express data flow relations

Pros

- fast
- complete
- always terminates (compare with unification)

Cons

- Imprecise -> Polymorphism

Cartesian Product Algorithm

- all function calls have a list of argument type sets
- take CP of that list
 - now you have a list of argument lists, but each argument is monomorphic
- make a new template for each list of monomorphic argument
- connect return value of all those templates together
- reuse templates

CPA (part 2)

Ideas

- assume nothing (empty type sets initially)
- monotonicity (only add info, never remove)
- iteration v. integration
 - solve mutually interdependent problems by
 - incrementally providing incomplete, but always correct info

Pros

- excellent precision
- efficient
 - only do exactly as much work as maximal precision requires

Cons

- data polymorphism still leads to imprecision
- no easy way to handle eval/exec

Why Python makes TI hard

- closures
- eval / exec
- multiple inheritance (depth first resolution)
- lots of C extension modules
 - need some way to specify the type behavior of foreign code

Harder Still . . .

- function/class definitions are imperative, not declarative
- introspection
 - instance.attribute equivalent to instance.__dict__['attribute']
 - instances, classes, modules, functions are real dictionaries

Still Harder . . .

no declarations & no access control

- variables & attributes created dynamically via assignment
- anyone can add attributes to an instance or class
- (can add/change class method and all instances see the change)
- requires whole program analysis

II. Starkiller

- Goals
- Design
- New Ideas

Goals

- Provide info for efficient native code compilation
 - Enable pay for what you use
 - Better performance than C/C++/Java
 - CPA can statically resolve dynamic dispatch
 - Dyn dispatch destroys many optimization techr (no static control graph)
- Catch runtime errors at compile time

Design

Think in terms of Two Graphs:

- Lexical Tree
 - describes definition relationship between prog entities
- Type annotated Control Graph

Being Insensitive

Flow Insensitive: assign types to variables

Flow Sensitive: assign types to variable accesses

Being insensitive is so easy. . .

Sensitivity requires

- transforming input into single static assignment for, or
- computing set of "reachable definitions" visible at every access

Cost/Benefit

- Agesen saw minimal benefit adding SSA
- Its unclear how to be sensitive outside code blocks

OOPs!

Classes and instances are first class objects, so

- create a new template for each class
- and one shared by all instances of that class
- instance types include a copy of their class type
- template stores attribute info
- functions handled specially
 - (not at define time since they can be added dynamically)
- getattr/setattr invoked on a class type connects to the class template
- setattr invoked on an instance type connects to the instance template
- getattr invoked on an instance type connects to both the instance and class templates.
- data polymorphism has removed my will to live

Problem: Lexical Scoping

Agesen's handling of lexical scoping had issues

- create a lexical pointer for each closure that refers to enclosing template
- never reuse closures
- complex, hard to understand, slow
- recursive customization means that analysis of recurs functions that pass closures may never terminate
 - no way around this except heuristics

New Idea: Lexical Scoping

Agesen's handling of lexical scoping invented a new method for handling nonlocal dependencies

Lets just use CPA to handle them

- Out of scope references are detected statically
- Each such reference becomes a silent argument for the function
- Definitions spit out a CP of the function/class type based on its argument types
- Each function/class type now contains the type info needed for its lexical children
- At call time, argument types are supplied

III. Challenges & Future Work

- Data Polymorphism
- Narcissism (introspection problems)
- Rollback
- Incremental Analysis
- Force Feedback
- Partial Evaluation
- Output Questions

Data Polymorphism

Some ideas

- Make a new instance template for each constructor call
 - yuck! expansion algorithms suck!
- Use indirection
 - every constructor call generates a new instance, but
 - instance states are cached
 - any method that changes instance state causes that instance "promoted" up to the new state
 - instance states are shared

Data Polymorphism (part 2)

More ideas . . .

- Be more CPA like
 - method calls should perform CPA on attribute of instances
 - method templates get shared
- Determine what attributes a method depends on or use those in CPA

Narcissism

Introspection Problem

- `foo.__dict__`, `getattr(foo, ...)`, `setattr(foo, ...)`
- objects can be treated like dictionaries

Implementation idea

- treat objects as dictionaries
- precision will be terrible b/c initial dictionary support merges to all dict reads and writes
- common case for objects is to have only statically known attrik accesses
- make dicts support that by making set/get dependant on value
- make constant values propogate by tagging them at their source removing the tag at function arg sites

Rollback

Allow propogations to be rolled back, "undone"

- tag types with their path through the net
- remove type from each stage
- propogate from all nodes that connect to nodes that h removed
- cycles? termination?

Couple this with template garbage collection

- this leaves unaccessibly nodes lying around
- need template gc anyway since some templates are c that shouldn't exist even in standard CPA

Incremental Analysis

Since python required whole program analysis

- incremental compilation is needed to make TI fast enough for development

Like this

- serialize constraint network and nodes + code objects
- code references include a hash of the code
- at analysis, compare code hashes to existing ones
- for different hashes, remove (rollback) all templates
- analyze new code and add templates as before.

Force Feedback

Imprecision isn't going away

Sometimes I'm reckless

- when the user doesn't care about maximal safety, get better inf Microsoft Quality Standards

Like this

- assume that any propogation that would normally trigger a warning/error will never happen
- rollback that propogation

More precise inference, although it may no longer be complete.
optimization on well tested code

Partial Evaluation

Improve precision by ignoring code paths that can't happen

What info is available at TI time?

- constants
- type info! you can determine the value of these at TI time
 - `foo.__class__`
 - `type(foo)`
 - `isinstance(x)`, etc.

Requires aggressive constant folding and propagation
Does that require flow sensitive analysis?

Narcissim 2: PE to the rescue

This also helps with Introspection problems

- people rarely use `__dict__` to store completely (statically) unknown keys
- they often do this:
 - `(getattr(self, 'handle_' + foo.name))(foo)`
- keys are often partially known statically and known to a limited form and from a statically known limited set of values.

Output?

How do I present info to user?

How do I display the data flow that triggered the error?

How do I explain complex types?

How do I avoid overwhelming the user, drowning them in
of useless information?