Design and Evaluation of Gradual Typing for Python

Michael M. Vitousek Andrew M. Kent Jeremy G. Siek
Jim Baker[†]

Indiana University Bloomington [†]Rackspace, Inc.

October 21, 2014

Gradual typing

```
def abs(n: int)->int:
  if n < 0:
    return -n
  else:
    return n

def dist(x, y):
  return abs(x - y)
```

Gradual typing

```
def abs(n: int)->int:
  if n < 0:
    return -n
  else:
    return n

def dist(x:Dyn, y:Dyn)->Dyn:
  return abs(x - y)
```

Gradual typing

```
def abs(n: int)->int:
  if n < 0:
    return -n
  else:
    return n

def dist(x:Dyn, y:Dyn)->Dyn:
  return abs(cast(x - y, Dyn, int))
```

Gradual typing for Python

Challenges

- Static type system
- Semantics of runtime checks
- Evolution of code from dynamic to static

Gradual typing for Python

Challenges

- Static type system
- Semantics of runtime checks
- Evolution of code from dynamic to static

Today, focus on runtime checks

Gradual typing for Python

Challenges

- Static type system
- Semantics of runtime checks
- Evolution of code from dynamic to static

Today, focus on runtime checks
Need to experiment!

Reticulated Python

Reticulated Python

- Gradual typing for Python
- Typechecker and source-to-source translator
- Python libraries that implement runtime semantics
- Testbed for different dialects
    - Common static semantics
    - Different dynamic semantics

Case studies

- CherryPy: web application framework
  - Heavy use of objects
  - Some APIs annotated, most of the code left as-is
  - 40k LoC
- Statistics library
  - Functions, lists, math, and `eval`
  - Annotated functions wherever possible
  - 1250 LoC
- SlowSHA
  - Objects and math
  - Annotated functions and objects
  - 350 LoC
- Unmodified Python Standard Library
  - 150k LoC

Bugs in target programs

```
class Node:
  def appendChild(self, node):
    ...
class Entity(Node): # subclass of Node
  def appendChild(self, newChild):
    ...
```

Python allows keyword arguments to be used anywhere:

`n.appendChild(node=Node())`

If `n` is an `Entity`, call will fail

Problem: object casts

Designing semantics for casts

- Casts on basic values
    - Just asserts

Designing semantics for casts

- Casts on basic values
  - Just asserts

```
def cast(v, t1, t2):
  if t1 == Dyn and t2 == int:
    assert isinstance(v, int)
    return v
```

Designing semantics for casts

- Casts on basic values
  - Just asserts

```
def cast(v, t1, t2):
  if t1 == Dyn and t2 == int:
    assert isinstance(v, int)
    return v
```

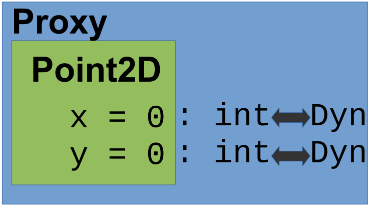
- Functions, other complex values harder

Designing semantics for casts

- Casts on basic values
  - Just asserts

```
def cast(v, t1, t2):
  if t1 == Dyn and t2 == int:
    assert isinstance(v, int)
    return v
```

- Functions, other complex values harder
  - Traditional approach: function wrappers and object proxies



Proxy

Point2D

`x = 0` : int↔Dyn

`y = 0` : int↔Dyn

Identity preservation is important

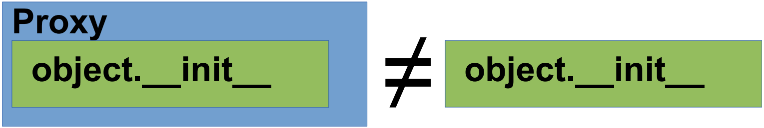
```
class _localbase(object):
  def __new__(cls, *args, **kw):
    if args or kw and\
        (cls.__init__ is object.__init__):
      raise TypeError("Initialization arguments are
          not supported")
```

- Under standard semantics, `cls.__init__` may be a proxy. . .

Identity preservation is important

```
class _localbase(object):
  def __new__(cls, *args, **kw):
    if args or kw and\
        (cls.__init__ is object.__init__):
      raise TypeError("Initialization arguments are
          not supported")
```

- Under standard semantics, `cls.__init__` may be a proxy. . .
    - And a proxy is *not* pointer-identical (`is` operator) to its underlying object



Proxy

object.__init__ ≠ **object.__init__**

- Python Standard Library relies on object identity. CherryPy unable to run

Designing casts for mutable objects

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Object({'x': int,'y': int}))->int:
  bad_update(pt)
  return pt.x
update_x(Point2D())
```

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(pt)
  return pt.x
update_x(Point2D())
```

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(pt)
  return pt.x
update_x(Point2D())
```

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```

Designing casts for mutable objects

- Traditional approach — *guarded* semantics
- Alternative approaches — *transient* and *monotonic* semantics
- Erasure semantics — no runtime checking whatsoever

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```

The *guarded* system

Casts install proxies that enforce typing

The *guarded* system

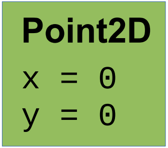
Casts install proxies that enforce typing

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```

The *guarded* system

Casts install proxies that enforce typing

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



Point2D

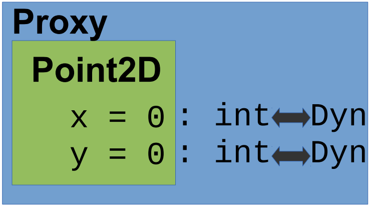
x = 0

y = 0

The *guarded* system

Casts install proxies that enforce typing

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



Proxy

Point2D

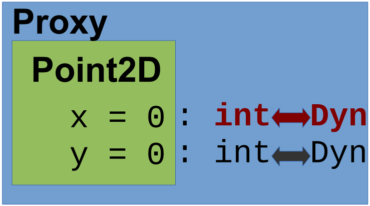
x = 0 : int⟷Dyn

y = 0 : int⟷Dyn

The *guarded* system

Casts install proxies that enforce typing

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



Proxy

Point2D

x = 0 : **int⟷Dyn**

y = 0 : int⟷Dyn

The *guarded* system

- Pros: well-understood, straightforward
- Cons: breaks object identity

	CherryPy	`stats.py`	SlowSHA
Guarded	✗	✓	✓
Transient			
Monotonic			

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

```
@fields({’x’: int, ’y’: int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = ’42’
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return check(pt.x, int)
update_x(Point2D())
```

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return check(pt.x, int)
update_x(Point2D())
```

Point2D

x = 0

y = 0

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return check(pt.x, int)
update_x(Point2D())
```

Point2D

x = '42'

y = 0

The *transient* system

Casts give only temporary guarantees, recover soundness with use-site checks

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return check(pt.x, int)
update_x(Point2D())
```

Point2D

x = '42'
y = 0

The *transient* system

- Casts only return the casted value itself (or an error), don't prevent future modification
- Checks ensure that later mutations will be detected (if they become relevant)
- Pros: Preserves object identity, simple to implement, *works*
- Cons: Many checks inserted, slightly more permissive, weaker invariants

	CherryPy	`stats.py`	SlowSHA
Guarded	✗	✓	✓
Transient	✓	✓	✓
Monotonic			

The *monotonic* system

Use RTTI to ensure that statically typed reads and writes don't go through casts

The *monotonic* system

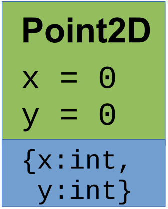
Use RTTI to ensure that statically typed reads and writes don’t go through casts

```
@fields({’x’: int, ’y’: int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = ’42’
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```

The *monotonic* system

Use RTTI to ensure that statically typed reads and writes don’t go through casts

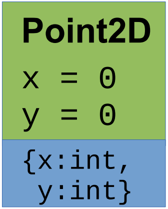
```
@fields({’x’: int, ’y’: int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = ’42’
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



The *monotonic* system

Use RTTI to ensure that statically typed reads and writes don't go through casts

```
@fields({'x': int, 'y': int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = '42'
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



Point2D

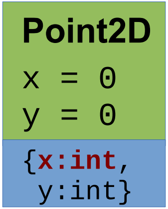
x = 0
y = 0

{x:int,
 y:int}

The *monotonic* system

Use RTTI to ensure that statically typed reads and writes don’t go through casts

```
@fields({’x’: int, ’y’: int})
class Point2D:
  x = 0
  y = 0

def bad_update(pt):
  pt.x = ’42’
def update_x(pt:Point2D)->int:
  bad_update(cast(pt,Point2D,Dyn))
  return pt.x
update_x(Point2D())
```



Point2D

x = 0

y = 0

{x:int,
 y:int}

The *transient* system

- RTTI becomes monotonically less dynamic as objects flow through casts
- Pros: No casts on reads from statically-typed code, preserves object identity
- Cons: RTTI is permanent and casts cause action-at-a-distance
- Under construction

	CherryPy	`stats.py`	SlowSHA
Guarded	✗	✓	✓
Transient	✓	✓	✓
Monotonic			✓

Conclusions

- Reticulated Python is a framework for experimenting with gradual typing in Python
- Performed case studies on real Python code
- Found bugs
- Implemented and tested semantics designs
    - Found that object identity problems from proxies are a major practical issue
    - Showed that transient is a reasonable alternative
- Lots more: static type system, type inference, implementation of the runtime semantics, load-time typechecking

Check it out: `github.com/mvitousek/reticulated`

Thanks!