Gradual Typing for Python

Jeremy Siek and Joe Angell
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder
Gradual Typing

- Static and dynamic type systems have complimentary strengths.
- Static typing provides full-coverage error checking, efficient execution, and machine-checked documentation.
- Dynamic typing enables rapid development and fast adaption to changing requirements.
- Why not have both in the same language?
Goals for gradual typing

• Treat programs without type annotations as dynamically typed.
• Programmers may incrementally add type annotations to gradually increase static checking.
• Annotate all parameters and the type system catches all type errors.
• The type system and semantics should place a minimal implementation burden on language implementors.
Implicit coercions to/from dynamic

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx
a = 1
p = Point()
p.move(a)
```

Parameters with no type annotation are given the dynamic type.
Implicit coercions to/from dynamic

class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)

Parameters with no type annotation are given the dynamic type.
Implicit coercions to/from dynamic

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
```

Parameters with no type annotation are given the dynamic type.
Implicit coercions to/from dynamic

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
```

Parameters with no type annotation are given the dynamic type.
Implicit coercions to/from dynamic

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
```

Parameters with no type annotation are given the dynamic type.
Implicit coercions to/from dynamic

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
```

Parameters with no type annotation are given the dynamic type.
Detecting static type errors

class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
p.move("hi")
Detecting static type errors

```python
class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx
a = 1
p = Point()
p.move(a)
p.move("hi")
```
Detecting static type errors

class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

a = 1
p = Point()
p.move(a)
p.move("hi")
Detecting static type errors

class Point:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx
a = 1
p = Point()
p.move(a)
p.move("hi")
Type System Primer

Given an expression e, a type T, and a dictionary Γ that maps variables to types, the notation

\[ \Gamma \vdash e : T \]

roughly means

\[ T = \text{typecheck}(\Gamma, e) \]

and the horizontal bar notation:

\[ \begin{array}{c} P_1 \ \\ \hline \ \\ P_2 \ \\ \hline \ \\ P_3 \ \\ \hline \ \\ Q \end{array} \]

means

Q if P_1 and P_2 and P_3
Gradual Typing: replace equality with consistency (~)

\[ \Gamma \vdash e_1 : \text{object\{..., \text{m} : S \rightarrow T,\ldots\}} \]

\[ \Gamma \vdash e_2 : S' \quad S' \sim S \]

\[ \Gamma \vdash e_1.m(e_2) : T \]
Gradual Typing: replace equality with consistency (~)

\[
\begin{align*}
\Gamma \vdash e_1 & : \text{object}\{..., \ m : S \rightarrow T,\ldots\} \\
\Gamma \vdash e_2 & : S' \\
\end{align*}
\]
\[
\Gamma \vdash e_1.m(e_2) : T
\]
The consistency relation

- Definition: a type is **consistent**, written ~, with another type when they are equal in the places both are defined.

- Examples:

  - `int ~ int`
  - `int \not\sim bool`
  - `dyn ~ int`
  - `int ~ dyn`
  - `object{x:int \rightarrow dyn, y: dyn \rightarrow bool} ~ object{y:bool \rightarrow dyn, x:dyn \rightarrow bool}`
  - `object{x:int \rightarrow int, y:dyn \rightarrow bool} \not\sim object{y:dyn \rightarrow bool, x:bool \rightarrow int}`
  - `object{x:int \rightarrow int, y:dyn \rightarrow dyn} \not\sim object{x:int \rightarrow int}`
Consistency

dynamic ~ T

T ~ dynamic

\[ S_1 \sim T_1 \quad S_2 \sim T_2 \]

\[ S_1 \rightarrow S_2 \sim T_1 \rightarrow T_2 \]

dom(\Gamma_1) = dom(\Gamma_2)

for all x in dom(\Gamma_1). \( \Gamma_1(x) = T_1 \) and \( \Gamma_2(x) = T_2 \)

implies \( T_1 \sim T_2 \)

object\{\Gamma_1\} \sim object\{\Gamma_2\}
Gradual Typing for Python

Dynamic Typing

Gradual Typing

Static Typing

Note: type annotation syntax based on Python 3k
A Static Type System for Python

- Nominal vs. Structural Types
- Subtyping vs. Matching
- Generics with match bounds
Nominal vs. Structural

```python
class Thing1:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing1()

class Thing2:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing2()
```
Nominal vs. Structural

```python
class Thing1:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing1()
```

```python
class Thing2:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing2()
```
Nominal vs. Structural

```python
class Thing1:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing1()

class Thing2:
    def __init__(self):
        self.x = 0
    def move(self, dx : int):
        self.x = self.x + dx

p = Thing2()
```

```
object{move: int→none}
```

```
object{move: int→none}
```
Subtyping vs. Matching

• There’s two approaches to structural typing
• Structural Subtyping: thoroughly explored by Luca Cardelli and many others.
• Matching: invented by Kim Bruce, similar to OCaml’s approach to objects, less well explored.
The Problem with Subtyping: Binary Methods

class Point:
    def __init__(self : Point):
        self.x = 0.0; self.y = 0.0
    def equal(self : Point, other : Point) -> bool:
        return x == other.x and y == other.y

class ColorPoint(Point):
    def __init__(self : ColorPoint):
        Point.__init__(self)
        self.c  = 'red'
    def equal(self : ColorPoint, other : ColorPoint) -> bool:
        return Point.equal(self, other) and self.c == other.c
The Problem with Subtyping: Binary Methods

class Point:
    def __init__(self : Point):
        self.x = 0.0; self.y = 0.0
    def equal(self : Point, other : Point) -> bool:
        return x == other.x and y == other.y

class ColorPoint(Point):
    def __init__(self : ColorPoint):
        Point.__init__(self)
        self.c = 'red'
    def equal(self : ColorPoint, other : ColorPoint) -> bool:
        return Point.equal(self, other) and self.c == other.c

ColorPoint = object{__init__ : () -> ColorPoint, equal: ColorPoint -> bool}
The Problem with Subtyping: Binary Methods

class Point:
    def __init__(self : Point):
        self.x = 0.0; self.y = 0.0
    def equal(self : Point, other : Point) -> bool:
        return x == other.x and y == other.y

class ColorPoint(Point):
    def __init__(self : ColorPoint):
        Point.__init__(self)
        self.c = ‘red’
    def equal(self : ColorPoint, other : ColorPoint) -> bool:
        return Point.equal(self, other) and self.c == other.c

ColorPoint = object{__init__ : () -> ColorPoint, equal: ColorPoint -> bool}
The Problem with Subtyping:
Binary Methods

class Point:
    ...

class ColorPoint(Point):
    ....
        def equal(self : ColorPoint, p : Point) -> bool:
            other = dynamic_cast<|ColorPoint|>(p)
            return Point.equal(self, other) and self.c == other.c

class Point3D(Point):
    ...

p1 = ColorPoint()
p2 = Point3D()
p1.equal(p2) // no compile error, instead get run-time error, bad!
Matching & Binary Methods

```python
class Point:
    def __init__(self : selftype):
        self.x = 0.0; self.y = 0.0
    def equal(self : selftype, other : selftype) -> bool:
        return x == other.x and y == other.y

class ColorPoint(Point):
    def __init__(self : selftype):
        Point.__init__(self)
        self.c = 'red'
    def equal(self : selftype, other : selftype) -> bool:
        return Point.equal(self, other) and self.c == other.c

OK!
Look Ma, no dynamic cast!
```
Matching & Binary Methods

p1 = ColorPoint()
p2 = ColorPoint()
p1.equal(p2) // OK!

p3 = Point3D()
p1.equal(p3) // compile error, good!
Matching: Under the Hood

\[ \Gamma \vdash e : T \quad T \preceq \text{object}\{m : S\} \]

\[ \Gamma \vdash e.m : [\text{selftype:=}T]S \]
Matching + Consistency

\[ \text{dom(} \Gamma_2) \subseteq \text{dom(} \Gamma_1) \]

for all \( x \) in \( \text{dom(} \Gamma_2) \). \( \Gamma_2(x) = T_2 \) and \( \Gamma_1(x) = T_1 \)

implies \( T_1 \sim T_2 \)

\[ \text{object}\{\Gamma_1}\ <# \text{object}\{\Gamma_2}\]
Generics

- Lots of Python code is polymorphic
- Generics are needed to provide enough flexibility in the type system to handle this.
- (Of course, you can always fall back on using type dynamic when you want to.)
Generic Functions

def pow<| T |>(f : fun(T,T), n : int) -> fun(T,T):
    def pow_f(x : T):
        while n > 0:
            x = f(x); n -= 1
        return x
    return pow_f

def add1(x : int) -> int: return x + 1
add5 = pow<|int|>(add1, 5)
add5 = pow(add1, 5)
Generic Functions

def pow<T>(f : fun(T, T), n : int) -> fun(T, T):
    def pow_f(x : T):
        while n > 0:
            x = f(x); n -= 1
        return x
    return pow_f

def add1(x : int) -> int: return x + 1

add5 = pow<int>(add1, 5)
add5 = pow(add1, 5)
Generic Functions

def pow<! T >!(f : fun(T,T), n : int) -> fun(T,T):
    def pow_f(x : T):
        while n > 0:
            x = f(x); n -= 1
        return x
    return pow_f

def add1(x : int) -> int: return x + 1

add5 = pow<!int>!(add1, 5)
add5 = pow(add1, 5)

explicit instantiation
Generic Functions

```python
def pow<T>(f : fun(T,T), n : int) -> fun(T,T):
    def pow_f(x : T):
        while n > 0:
            x = f(x); n -= 1
        return x
    return pow_f

def add1(x : int) -> int: return x + 1
add5 = pow<int>(add1, 5)
add5 = pow(add1, 5)
```

**explicit instantiation**

**implicit instantiation**
Generic Classes and Methods

class Point<|T|>:
    def __init__(self, x : T, y : T):
        self.x = x; self.y = y

    def map<|U|>(self, f : fun(T,U)) -> Point<|U|>:
        return Point<|U|>(f(self.x), f(self.y))

>>> p = Point<|int|>(1, 3)
>>> p.map(float)
Point<|float|>(1.0, 2.0)
Generic Classes and Methods

class Point<T>:
    def __init__(self, x : T, y : T):
        self.x = x; self.y = y

    def map<U>(self, f : fun(T, U)) -> Point<U>:
        return Point<U>(f(self.x), f(self.y))

>>> p = Point<int>(1, 3)
>>> p.map(float)
Point<float>(1.0, 2.0)
**Generic Classes and Methods**

```python
class Point<|T|>:
    def __init__(self, x : T, y : T):
        self.x = x; self.y = y

    def map<|U|>(self, f : fun(T,U)) -> Point<|U|>:
        return Point<|U|>(f(self.x), f(self.y))

>>> p = Point<|int|>(1, 3)
>>> p.map(float)
Point<|float|>(1.0, 2.0)
```
**Match-bound Generics**

```python
from typing import TypeVar, Generic

typealias Iterator = forall(T, object({next: method(selftype, T)}))
typealias Iterable = forall(T, It (# Iterator<|T|>,
                                object({__iter__: method(selftype, It) })))

def mymap<| T, R, It (# Iterable<|T|> |> ( f : fun(T, R), l : It) -> list<|R|>:
    res = list<|R|>()
    for x in l: res.append(f(x))
    return res

def doubledown(a : int) -> int:
    return 2 * a

alist = [1,2,3]
mymap(doubledown, alist)
```
Match-bounds: Under the Hood

$\Gamma \vdash e : \forall (X_1<\#T_1, \ldots, X_n<\#T_n). R$

$S_1 <\# T_1, \ldots, S_n <\# T_n$

$\Gamma \vdash e<|S_1,\ldots,S_n|> : [X_1:=S_1,\ldots, X_n:=S_n] R$
Future Work

- Public release of the type checker
- Corpus analysis to test whether our type system is a good fit for Python programs
- Integration with Jython
- Implement run-time type checks
- Compiler optimizations to take advantage of the type information provided by gradual typing
Conclusion

- Gradual typing integrates static and dynamic typing in the same language, based on a new relation on types called consistency.

- For Python, we hope a type system based on structural matching and generics will provide a good fit.

- To see a demo, find Joe during a break, or better yet, over a beer!
Why add types to a dynamic language?

- Large applications suffer from “dynamicitis”
- Values of many different kinds flow through the same point within the system. Checking code appears everywhere.
- E.g., in Django, there are places where True, ‘t’, ’T’, ‘True’, and 1 are all potential values that must be dealt with
- Annotate interfaces with types to establish invariants and document expectations
Why not ML-style inference?

- Inference alone does not provide the flexibility of dynamic typing.
- Combining inference and gradual typing may provide the best of both worlds.
- See *Gradual Typing and Unification-based Inference* by Siek and Vachharajani.