Gradual Types

CS242

Lecture 14

Flashback …

- There is a split in the world of programming between
  - Typed languages
  - Untyped languages

- Leave the religious debate aside for now ...
  - We'll come back to why there is a debate at all

Statically vs. Dynamically Typed Languages

- Lecture outline

- Brief review of static typing
  - and its tradeoffs
- Dynamic typing
  - and its tradeoffs
- Gradual typing in MyPy
  - an example of the kind of compromise that is gaining popularity

Today's Lambda Calculus Variation …

$e \rightarrow x \mid \lambda x{:}t.e \mid e\ e \mid i \mid e + e$

Static Type Rules

$$\frac{}{A, x{:}\, t \vdash x : t} \quad \text{[Var]}$$

$$\frac{A, x{:}\, t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \rightarrow t'} \quad \text{[Abs]}$$

$$\frac{}{A \vdash i : \text{Int}} \quad \text{[Int]}$$

$$\frac{A \vdash e_1 : t \rightarrow t' \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'} \quad \text{[App]}$$

$$\frac{A \vdash e_1 : \text{Int} \qquad A \vdash e_2 : \text{Int}}{A \vdash e_1 + e_2 : \text{Int}} \quad \text{[Add]}$$

Benefits of Static Typing

- Find type errors when the program is written
  - Well, actually when the type checker runs

- Makes guarantees for *all* possible executions
  - For all time, no matter the input

- Faster execution
  - No need to do any runtime type checking
  - The types have already been checked ...

Dynamic Typing

- Instead of checking types at compile time, types are checked when the program executes

- Sound: Type errors during execution will be detected

- Not the same as untyped languages!

How Does Dynamic Typing Work?

- In the implementation, *every* value has a "tag" indicating its type

- In our simple language, we need two tags: int and fun

- Note that the fun tag says only "this is a function"
  - No information about the domain and range types

- When an operation requires a certain type, we check the tag
  - And remove it to get the untagged value

Dynamically Typed Lambda Calculus

The language we write programs in is the same (w/o types):

e → x | λx.e | e e | i | e + e

We *complete* programs to make the dynamic checks explicit

- !t e   mark e with the type tag t
- ?t e   check if e has the type tag t (and remove the tag)

e → x | λx.e | e e | i | e + e | !t e | ?t e

t → int | fun

Dynamic Typing Program Translation Rules

$$\frac{}{x \hookrightarrow x}\ [\text{Var}]$$

$$\frac{e \hookrightarrow e'}{\lambda x.e \hookrightarrow\ !\text{fun}\ \lambda x.e'}\ [\text{Abs}]$$

$$\frac{}{i \hookrightarrow\ !\text{int}\ i}\ [\text{Int}]$$

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{e_1 + e_2 \hookrightarrow\ !\text{int}\ ((?\text{int}\ e_1') + (?\text{int}\ e_2'))}\ [\text{Add}]$$

$$\frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{e_1\ e_2 \hookrightarrow (?\text{fun}\ e_1')\ e_2'}\ [\text{App}]$$

Example

$(\lambda x.x + 1)\ 2 \ \hookrightarrow$?fun (!fun (λx. int! (int? x + int? (int! 1))) int! 2

Fully parenthesized:

$(\lambda x.x + 1)\ 2 \ \hookrightarrow$ (?fun (!fun (λx. int! ((int? x) + (int? (int! 1)))))) (int! 2)

Rules for Execution

$$\frac{E \vdash \lambda x.e \to \lambda x.e}{E \vdash \text{!fun } \lambda x.e \to \text{!fun } \lambda x.e} \text{[Box-Fun]}$$

$$\frac{E \vdash e \to \text{?fun (!fun } \lambda x.e'\text{)}}{E \vdash e \to \lambda x.e'} \text{[Unbox-Fun]}$$

$$\frac{E \vdash e \to i}{E \vdash \text{!int } e \to \text{!int } i} \text{[Box-Int]}$$

$$\frac{E \vdash e \to \text{?int (!int } i\text{)}}{E \vdash e \to i} \text{[Unbox-Int]}$$

Example

$(\lambda x.x + 1)\ 2 \hookrightarrow (?fun\ (!fun\ (\lambda x.\ !int\ (?int\ x + ?int\ (!int\ 1))))\ (!int\ 2)$

$\rightarrow (\lambda x.\ !int\ (?int\ x + ?int\ (!int\ 1)))\ (!int\ 2)$

$\rightarrow\ !int\ (?int\ (!int\ 2) + ?int\ (!int\ 1)))$

$\rightarrow\ !int\ (2 + ?int\ (!int\ 1))$

$\rightarrow\ !int\ (2 + 1)$

$\rightarrow\ !int\ 3$

Benefits of Dynamic Typing

- Guarantees that a specific program execution is type correct

- Easy to put together arbitrary pieces of code
  - No type checker nagging you!

- Lower barrier to entry for programmers
  - Don’t need to understand the (often complicated) type system
  - Only need to debug real errors, not obscure type checking error messages

Static vs. Dynamic Typing

Safety

Proving type safety of all executions vs. one execution

Productivity

Can't write some programs vs. can write all programs

Much to learn vs. little to learn

Performance

Faster code vs. Slower code

Slow compiles vs. No compiles

Observations

- The benefits of static typing grow with
  - The size of the program
  - The number of engineers involved
  - The size of the user community of the program

- Because
  - It is easier to reason by hand about a small program than a big one
  - Inconsistencies are inevitable when multiple people work on a project
  - Users don’t want to deal with developers’ bugs

Prediction

- So we expect big, performance-oriented programs to be written in statically typed languages
  - Operating systems, compilers, databases, web servers, numerical libraries, ...

- Where programs are small, performance is not important, or we need to compose programs in ways that static type systems cannot handle, we expect to see more use of dynamically typed languages
  - Scripts, client-side web programming

Reality

Dropbox has deployed more than four million lines of Python code and is one of a growing number of companies that annotate code written in the dynamic programming language to make it easier to debug and understand.

But PHP hasn't gone away–Facebook and other big organizations and projects have millions of lines of code written in the language, and programmers still appreciate it for rapid development and deployment, even as they try to steer clear of its messier features.

What Happened?

- A few things ...
- Small programs grew into big programs
- Network effect
  - Easier to write all the code for a system in fewer languages
- Training effect
  - Much easier to hire python programmers than C++ programmers

Reality

Dropbox has deployed more than four million lines of Python code and is one of a growing number of companies that annotate code written in the dynamic programming language to make it easier to debug and understand.

But PHP hasn't gone away–Facebook and other big organizations and projects have millions of lines of code written in the language, and programmers still appreciate it for rapid development and deployment, even as they try to steer clear of its messier features.

The Grass is Greener ...

- Large systems written in dynamically typed languages inevitably
  - Suffer from poor performance
  - And runtime type errors

- And, just as inevitably, there are efforts to convert the code base
  - Find a way to add type information
    - Python annotations
  - Build tools to try to do best effort type inference and optimize code
    - Facebook's PHP -> C++ compiler

Gradual Types

- There have been many efforts to integrate static type features into dynamically typed languages
  - Not much interest in the other direction

- *Gradual types* is one approach

The Dream

- We want a seamless transition from untyped to fully typed code
- A program with no types behaves as a dynamically typed program
- A program with all types filled in is statically typed
- And anything in between ...

MyPy

- A gradual type system for Python
  - Referred to as “an optional type system”

``Migrate existing code to static typing, a function at a time. You can freely mix static and dynamic typing within a program …’’

Features

- Type declarations
- Type inference
- The Any type
- Class types
- Subtyping
- Covariance, Invariance, Contravariance

Type Declarations

```
age: int = 1
tolerance: float = 0.3
x: list[int] = [1,2,3]
s: set[int] = {4,5}
d: dict[str,float] = {“Amazon River”: 3977.0}
```

Type for variables can be explicitly declared

- And will be checked by MyPy

Type Inference

```
def nums_below(numbers: Iterable[float], limit: float) -> list[float]:
    output = []
    for num in numbers:
        if num < limit:
            output.append(num)
    return output
```

Type Any

```
n = 1          # n inferred to have type Int
n = 'x'        # type error!

a: Any = 1
a = 'x'        # no type error
n = a          # no type error
n = n + 1      # runtime type error
```

Type Any, Cont.

- In type checking, Any matches any other type
- Another view: Any type error involving Any is suppressed
- Using values with dynamic type Any is the way to mix statically typed and dynamically typed code

Classes

```
Class Point:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def move(self,newx,newy):
       self.x = newx
       self.y = newy

  def getx(self):
       return self.x
  def(gety(self):
       return self.y
```

Inheritance

```
class ColorPoint(Point):
        def set_color(newc):
            self.color = newc
```

Nominal Subtyping

- ColorPoint is a subtype of Point
  - Because ColorPoint explicitly inherits from Point
  - This is *nominal subtyping*
  - By far the most common form of subtyping in practice

- We write ColorPoint $\leq$ Point
  - Anywhere a Point is expected, a ColorPoint can also be used

- Subtyping has some unexpected properties in static typing …

Structural Subtyping

- Python also supports *structural subtyping*
  - Also called *"duck typing"*

- We consider $X \leq Y$ because X implements the methods of Y
  - No explicit declaration that X inherits Y is needed, the compatibility is determined automatically

- Example: If a class defines a method __iter__ with suitable arguments, MyPy understands it is of type Iterable[T]

Subtyping

- Once it is determined that $A \leq B$, the use of subtyping is the same
  - Whether it is nominal or structural

- Subtyping has some interesting properties in static typing …

Example

```
def reset_point(p: Point):
        p.move(0,0)

c: ColorPoint = ColorPoint()
reset_point(c)
```

The Type Object

- Object is the root of the class hierarchy
  - C $\leq$ Object for any class C

- Note that Object is not the same as Any!
  - Object is itself a class
  - Any is a static type for any dynamically typed value

Covariant Typing

- Recall: Type constructors are functions that take types as arguments and return types as results
- A type constructor $C[A]$ is covariant if $X \leq Y$ implies $C[X] \leq C[Y]$
- Example: $X \leq$ Union[X,Y]

```
p: Union[Point,Int]
p = Point()
p = 1
```

Is List Covariant?

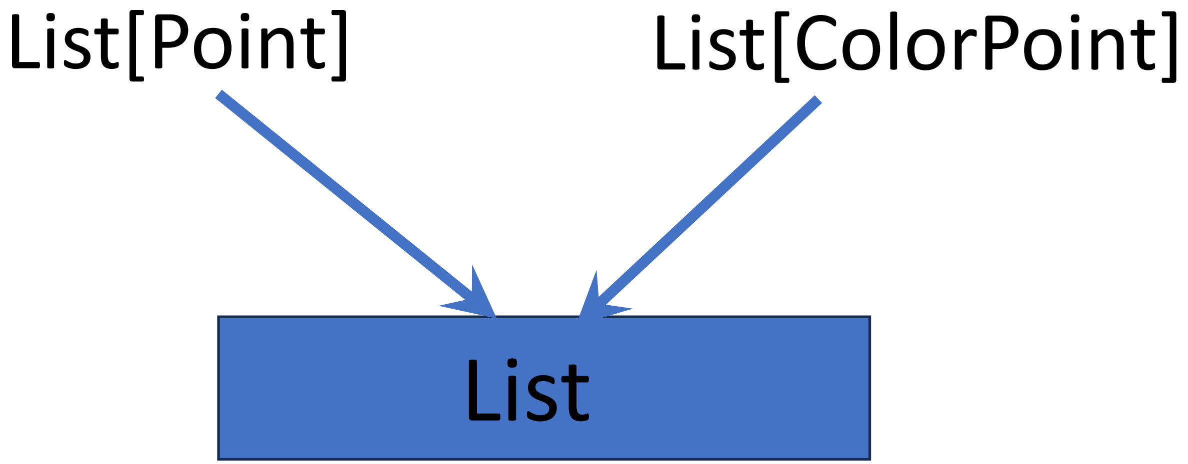
```
def addone(l: List[Point]) -> None:
        l.append(Point())

cp: List[ColorPoint] : []
addone(cp)                    # note ColorPoint <= Point
cp[0].set_color("blue")       # the Point has no set_color method!
```

What Went Wrong?

These two names are aliases of the same list in memory. But they have different types!

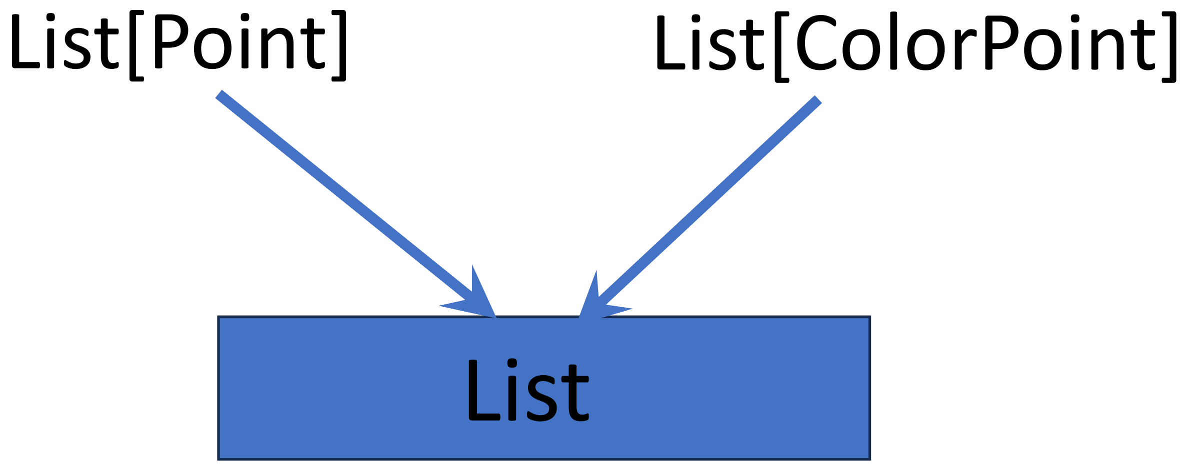
```
def addone(l: List[Point]) -> None:
        l.append(Point())

cp: List[ColorPoint] : []
addone(cp)                  # note ColorPoint ≤ Point
cp[0].set_color("blue")     # the Point has no set_color method!
```

Aliasing + Mutability + Covariance = Unsound!



If List is a mutable container, we can add an object of the supertype through one pointer, and call a method of the subtype on the other pointer.

Aliasing + Mutability + Covariance = Unsound!



Another explanation: Aliasing is a symmetric property, either pointer can be used to read/write the list. But subtyping is asymmetric. No matter which direction the subtyping goes, it will be unsound.

Invariant Typing

Subtyping cannot be used with mutable containers: They must use invariant typing.

```
def addone(l: List[Point]) -> None:
        l.append(Point())

cp: List[ColorPoint] : []
addone(cp)                    # type error List[Point] != List[ColorPoint]
cp[0].set_color("blue")       # the Point has no set_color method!
```

Contravariance

- Function types are contravariant in the domain, covariant in the range

- Example: $\text{Point} \rightarrow \text{Int} \leq \text{ColorPoint} \rightarrow \text{Int}$
  - If we expect a function that takes ColorPoint objects as arguments, it is OK to use a function that takes a larger class of arguments such as Point

- In general $A \rightarrow B \leq C \rightarrow D$ if $C \leq A$ and $B \leq D$

Example

```
def convertToInt(c: ColorPoint, f: Callable[[ColorPoint],Int]) -> Int:
    return f(c)

def getX(p: Point) -> Int:
     return p.x

convertToInt(ColorPoint(),getx)
```

Variance

- Most subtyping is covariant

- For mutable container types invariant typing is necessary for soundness
  - Because using subtyping will always be unsound for a pair of aliases of different types for some combination of writes through one and reads through the other

- Function subtyping is naturally contravariant in the domain

Summary

- Gradual typing allows dynamically typed and statically typed code to be mixed
  - In particular, allows the gradual addition of static types to otherwise dynamically typed code

- MyPy is the most popular gradual type system in use today
  - Includes subtyping to handle Python's class system
  - Includes generic type constructors to handle container types such as lists