Gradual Types

CS242

Lecture 12

Projects

There are three projects:

- Program verification using dependent types
- Gradual typing: Integrating static and dynamic type checking
- Writing an async library using Rust's novel features

- Similar in style to the homeworks
  - But bigger ... with sequential dependencies ... start early!

- Two deadlines!
  - Declare which project you will do (Monday, March 1)
  - Project due date (Friday, March 19)

Flashback to Lecture 4

- 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
  - 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

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

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

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

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

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

Benefits of Static Typing

- Find type errors when the program is written
  - Well, actually when you run the type checker
- 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):

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid i \mid e + e$

We rewrite 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

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid i \mid e + e \mid\ !t\ e \mid\ ?t\ e$

$t \rightarrow \text{int} \mid \text{fun}$

Dynamic Typing Program Translation Rules

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

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

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

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

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

Example

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

Fully parenthesized:

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

Rules for Execution

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

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

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

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

Example

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

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
  - The size of the user community

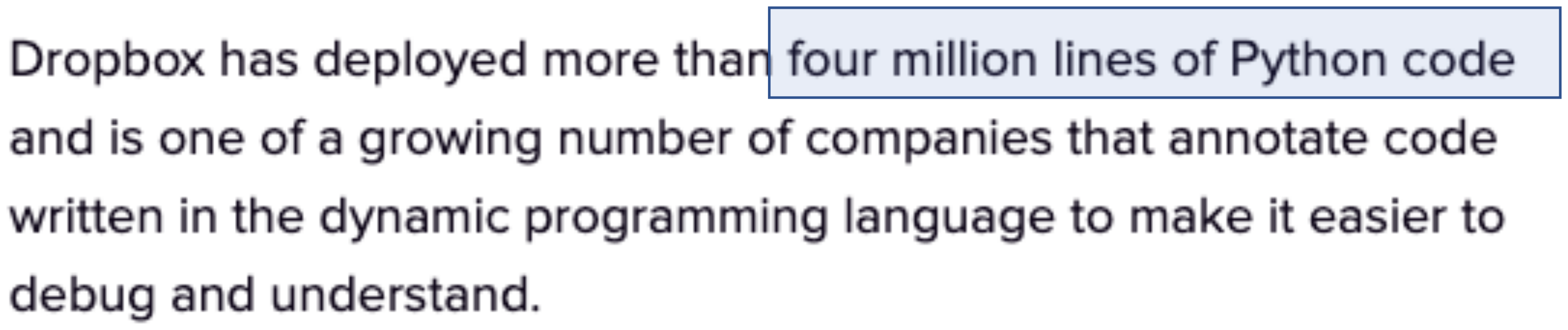
- Because
  - It is easier to reason by hand about a small program than a big one
  - Inconsistencies are inevitable when multiple people work on a project
  - Users don’t want to deal with developers’ bugs

Prediction

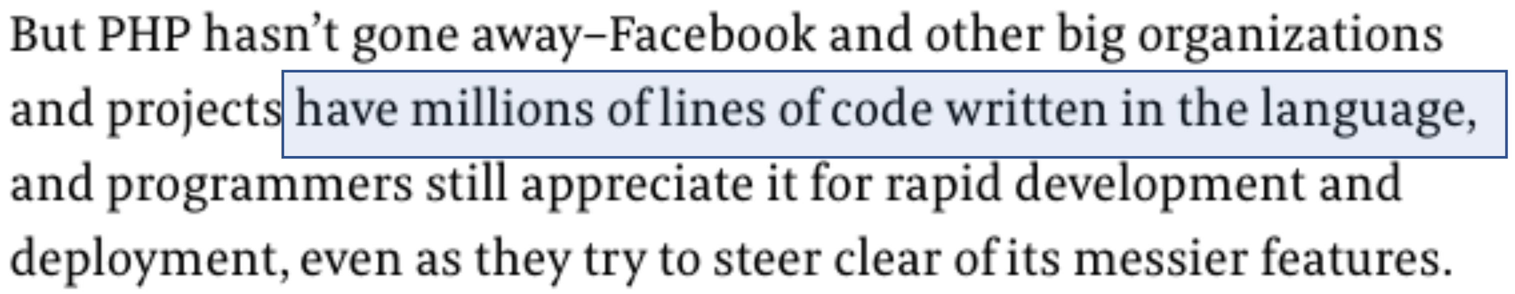
- So we expect big, performance-oriented programs to be written in statically typed languages
  - Operating systems, compilers, databases, web servers, numerical libraries, ...

- 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?

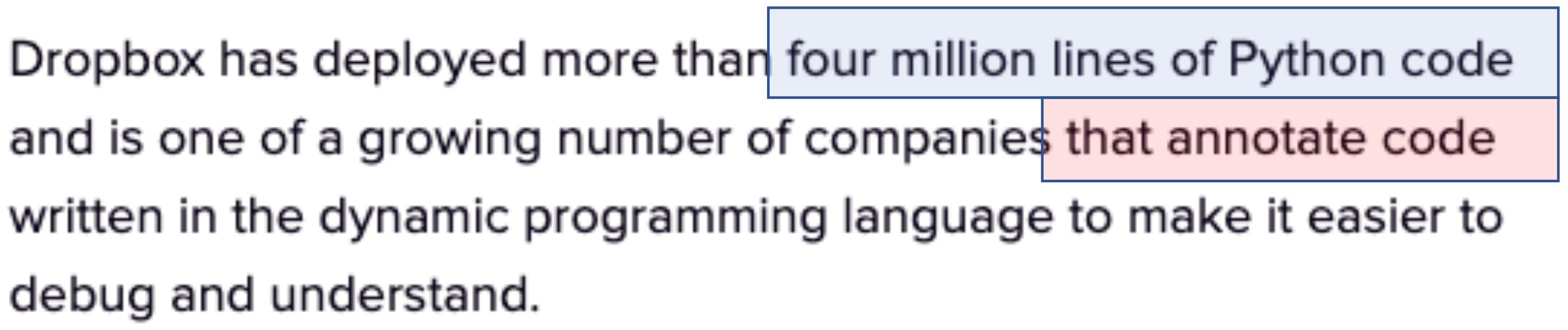
- A few things ...

- Small programs grew into big programs

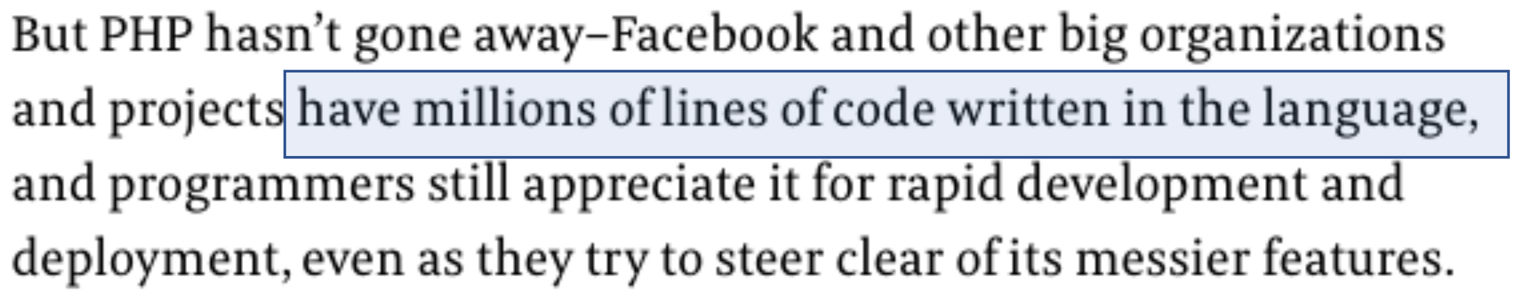
- Network effect
  - Easier to write all the code for a system in fewer languages

- 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 iinterest 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 ...

Gradual Types: The Idea

- Types themselves can vary between dynamic and fully static

• ?	*A dynamically typed (tagged) value*
• int → int	*Functions from integers to integers*
• int → ?	*Functions from integers to a dynamically typed value*
• ? → int	*Functions from a dynamically typed value to an integer*
• ? → ?	*Functions from dynamically typed values to d.t.v.*

Grammar

Programs can have type declarations:

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

Types can be gradual:

$t \rightarrow \alpha \mid t \rightarrow t \mid ?$

Gradual Type Rules

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

$$\frac{A \vdash e_1 : ? \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : ?}\ [\text{App1}]$$

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

$$\frac{A \vdash e_1 : t_1 \rightarrow t_3 \qquad A \vdash e_2 : t_2 \qquad t_1 \sim t_2}{A \vdash e_1\, e_2 : t_3}\ [\text{App2}]$$

Consistency Relation

Intuition: Given the information we have, the types could be equal

$t \sim t$ *Reflexive*

$? \sim t$ *Any type is consistent with type dynamic*

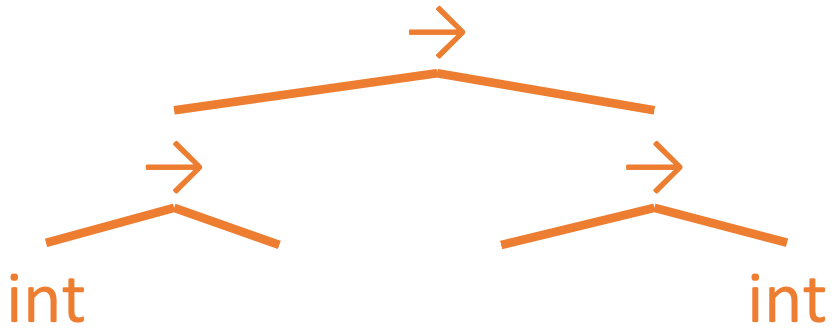
$t \sim ?$ *Any type is consistent with type dynamic*

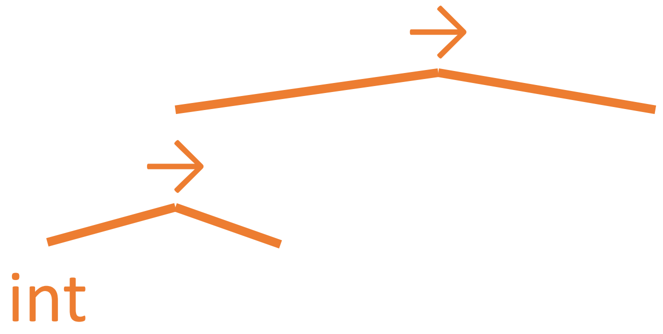
$t_1 \to t_2 \sim t_3 \to t_4$ iff $t_1 \sim t_3$ and $t_2 \sim t_4$

Two function types are consistent if their components are consistent

Example

$(\text{int} \to ?) \to (? \to \text{int}) \sim (\text{int} \to ?) \to ?$





Examples of Inconsistency

$\text{int} \not\sim\ ? \to ?$

$\text{int} \to\ ? \not\sim (\text{int} \to ?) \to ?$

Static + Dynamic Checking

- Static type rules are not enough for gradual types

- We are also doing dynamic checks
  - So we need a translation step that inserts those checks

- The checks are more complicated now
  - Not just tags
  - But entire types ...
  - Write the checks as *type casts*

Grammar

Programs can have type declarations:

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

Types can be gradual:

$t \rightarrow \alpha \mid t \rightarrow t \mid ?$

Translated terms:

$e \rightarrow x \mid \lambda x{:}\, t.e \mid e\ e \mid \langle t \rangle\, e$

Gradual Type and Translation Rules

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

$$\frac{A \vdash e_1 \hookrightarrow e_1' : ? \qquad A \vdash e_2 \hookrightarrow e_2' : t}{A \vdash e_1\, e_2 \hookrightarrow (\langle t \rightarrow ?\rangle\, e_1')\, e_2' : ?}\ [\text{App1}]$$

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

$$\frac{A \vdash e_1 \hookrightarrow e_1' : t_1 \rightarrow t_3 \qquad A \vdash e_2 \hookrightarrow e_2' : t_2 \qquad t_1 \sim t_2}{A \vdash e_1\, e_2 \hookrightarrow e_1'\, (\langle t_1\rangle\, e_2'): t_3}\ [\text{App2}]$$

The Last Detail …

- How do we implement these type casts?
  - No longer just tagged "this is a function"
  - Can't look inside a function to see what its domain and range are
  - Can only run the function

- So that is what we do
  - When a function *value* is cast to a function type

    <t → t'> λx:s.e

  - We know it is a function, so now rewrite it

    λz:t. <t'> (λx:s.e) (<s> z)

  - Function types are decomposed until we reach primitive types

- Other cast rules: <int> <?> i = i and <int> i = i
- Inconsistent casts such as <int> λx:s.e generate an error

An Example

- Untyped lambda calculus: $(\lambda f.f\ 1)\ (\lambda x.x)$
- With some gradual types: $(\lambda f{:}\ ? \rightarrow int.f\ 1)\ (\lambda x{:}int.x)$
- Translated with casts: $(\lambda f{:}\ ? \rightarrow int.f\ (\langle ? \rangle\ 1))\ \langle\ ? \rightarrow int \rangle\ (\lambda x{:}int.x)$

Execution

(λf: ? → int.f (<?> 1)) < ? → int> (λx:int.x)	*initial term*
(λf: ? → int.f (<?> 1)) λz:?. <int> ((λx:int.x) <int> z)	*function cast*
(λz:?. <int> ((λx:int.x) <int> z)) (<?> 1)	*application*
<int> ((λx:int.x) <int> (<?> 1))	*tagged integer cast*
<int> ((λx:int.x) 1)	*application*
<int> 1	*untagged integer cast*
1	

Summary

- There is a lot of interest in combining static and dynamic typing
  - And there is some interest in practice as well
- Gradual typing is not the only solution
  - Is one of the more popular
- Gives flexibility of dynamic typing
- And “pay as you go” static typing