Type systems

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Issues with Python
- Issues with Python and possible solutions
- Static typing

Issues with Python

Issues with Python

Lack of...

- Lack of constraints: how can we specify that a function only takes integers as input
- Lack of structure: how can we specify that a variable will certainly support some methods
- Lack of assurances: how can we guarantee that programs with evident errors are not run

Issues with Python

What is wrong with this?

```
def f(x):
  return (x * 2)
f(``nonsense'')
```

Issues with Python

What is wrong with this?

```
def f(x):
  return (x * 2)
f(''nonsense'')
```

The function clearly works with integers, but is given a string

Issues with Python

What is wrong with this?

```
x = input()
if (x > 100):
  print(‘‘dumb’’)
else:
  print(‘‘dumber’’)
```

Issues with Python

What is wrong with this?

```
x = input()
if (x > 100):
  print(``dumb'')
else:
  print(``dumber'')
```

The comparison is nonsensical if `x` is not a number

HOGESCHOOL ROTTERDAM

Issues with Python

What is wrong with this?

```
def g(car):
  return car.drive(2)
g(-1)
```

Issues with Python

What is wrong with this?

```
def g(car):
  return car.drive(2)
g(-1)
```

We expect something with a `drive` method, but get an integer instead

Possible solutions

Possible solutions

Testing?

- Testing the program should be enough

Possible solutions

Testing?

- Testing the program should be enough
- Right?

Possible solutions

Testing?

- Testing the program should be enough
- Right?
- No. The number of possible execution paths is immense (order of billions), and each test only takes one.
- Testing can only guarantee presence of bugs, but not their absence!

Possible solutions

How many times would we need to test to be sure there is no error?

```
if (randint(0,100000) > 99999):
  g(-1)
else:
  g(mercedesSL500)
```

How many times would we need to test to be sure there is no error?

```
if (randint(0,100000) > 99999):
  g(-1)
else:
  g(mercedesSL500)
```

≥ 100000

Possible solutions

Testing?

- We want our programming languages to perform checks for us
- Clearly nonsensical programs should be rejected before we can even run them
- It is safer and easier to spend more time "talking" with the IDE than hoping to find all errors at runtime

Static typing

Static typing

Introduction

- The language verifies[a], before running code, that all variables are correctly used
- "Correctly used" means that they are guaranteed to support all operations used on them
- This is by far and large the most typical solution to increase safety and productivity

[a] By means of the **compiler**.

Static typing

What is static typing?

- When declaring a variable, we also specify what sort of data it will contain
- The **sort** of data contained is called **TYPE** of the variable
- Types can be either primitives (int, string, etc.), custom (classes), or compositions (functions, list of elements of a given type, etc.)

What is static typing?

- Especially in mainstream languages, the specification of the type of a variable is done by hand by the programmer
- In other languages (mostly functional languages like F#, Haskell, etc.) the type of variables is automatically guessed by the compiler
- In our case our programs will become a bit more verbose but better specified
- Still, static typing is not necessarily connected with verbosity

Static typing

A variable declaration in C# or Java is prefixed by the type of the variable

- `int x;` declares an integer variable
- `string s;` declares a string variable
- `float f;` declares a floating point variable
- ...

Static typing

```
def f(x):
  return (x * 2)
```

Becomes, typed:

What has improved and why?

Static typing

```
def f(x):
  return (x * 2)
```

Becomes, typed:

What has improved and why?

The second definition encodes information about what goes in and what comes out of the function

Static typing

Is this still possible to write (as it was in Python)?

Static typing

Is this still possible to write (as it was in Python)?

No: we get a compiler error because a string cannot be used where a number is expected

Static typing


```
x = input()
if (x > 100):
  print(‘‘dumb’’)
else:
  print(‘‘dumber’’)
```

Becomes, typed:

What has improved and why?

Static typing


```
x = input()
if (x > 100):
  print(``dumb'')
else:
  print(``dumber'')
```

Becomes, typed:

What has improved and why?

The variable declaration specifies what is allowed (and what is not) inside the variable.

Static typing


```
def g(car):
  return car.drive(2)
g(-1)
```

Becomes, typed:

What has improved and why?

Static typing


```
def g(car):
  return car.drive(2)
g(-1)
```

Becomes, typed:

What has improved and why?

The function declaration specifies that `car` is an instance of the `Car` class. We will thus get a compiler error.

Typing rules and semantic rules

Typing rules and semantic rules

How do we describe them?

- How do we describe such relations clearly?
- We use the so-called **typing rules**, which specify what may be done and what not
- Typing rules are quite intuitive: they state that if one or more premises are true, then the conclusion is true as well

Typing rules and semantic rules

$$\frac{A \wedge \ B}{C}$$

If A and B are true, then we can conclude C

Typing rules and semantic rules

$$\frac{\text{I wish to buy a pretty car} \wedge \text{ I have 120000 euros}}{\text{I buy a Mercedes SL500}}$$

How do we read this rule?

Typing rules and semantic rules

$$\frac{\text{I wish to buy a pretty car} \wedge \text{ I have 120000 euros}}{\text{I buy a Mercedes SL500}}$$

How do we read this rule?

If I have 120000 euros and I wish to buy a pretty car, then I buy a Mercedes SL500

Typing rules and semantic rules

$$\frac{\text{It is raining} \wedge \text{ I have my umbrella with me}}{\text{I open my umbrella}}$$

How do we read this rule?

Typing rules and semantic rules

$$\frac{\text{It is raining} \wedge \text{ I have my umbrella with me}}{\text{I open my umbrella}}$$

How do we read this rule?

If I have my umbrella with me, and it is raining, then I open my umbrella

Reading typing rules

Let us apply this machinery to programming languages

Typing rules and semantic rules

Reading typing rules

- Let us apply this machinery to programming languages
- We will effectively give the specification of a modern compiler
- This looks like a "broadly scoped" execution of the program, and it is indeed such
- This process is called type checking

Typing rules and semantic rules

Reading typing rules

- We want to specify this in the typing rule notation
- The typing rules manipulate a stack of declarations which we will call D
- Each typing rule will add or remove variable declarations and return the type of the current expression
- Instead of coupling each variable with its value, we couple it with its type

Typing rules and semantic rules

- The simplest typing rule is the one that finds a variable declaration
- A declaration adds to the declarations D the variable, connected with its type

$$\frac{}{\langle (\texttt{T v;}), D\rangle \rightarrow \langle \texttt{void}, D[v \mapsto T]\rangle}$$

Typing rules and semantic rules


```
int x = 10;
```

Declarations:

PC
1

```
int x = 10;
```

Declarations:

PC	x
2	int

Typing rules and semantic rules

- When we look the variable up, its type is whatever type was found connected to it in the declarations
- This does further nothing to the declarations
- Let's assume that $\mathtt{x}$ is a variable name

$$\frac{}{\langle \mathrm{x}, D\rangle \rightarrow \langle D[\mathrm{x}], D\rangle}$$

Typing rules and semantic rules


```
int x = 10;
x = (x + 5);
```

Declarations:

PC
1

Typing rules and semantic rules


```
int x = 10;
x = (x + 5);
```

Declarations:

PC	x
2	int

```
int x = 10;
x = (x + 5);
```

Declarations:

PC	x
3	int

Typing rules and semantic rules

- Another simple typing rule is the one that types a constant value
- It does nothing to the declarations
- Let's assume that `i` is an integer constant

$$\frac{}{\langle \mathtt{i}, D\rangle \rightarrow \langle \mathtt{int}, D\rangle}$$

- Let's assume that `f` is a floating point constant

$$\frac{}{\langle \texttt{f}, D\rangle \rightarrow \langle \texttt{float}, D\rangle}$$

- Let's assume that `s` is a string constant

$$\frac{}{\langle \mathtt{s}, D\rangle \rightarrow \langle \mathtt{string}, D\rangle}$$

Typing rules and semantic rules

- Let's assume that `b` is a boolean constant

$$\frac{}{\langle \texttt{b}, D\rangle \rightarrow \langle \texttt{bool}, D\rangle}$$

Reading typing rules

More complex typing rules compose together the types of different statements

Typing rules and semantic rules

- The typing rule for operators such as + requires the operands to be compatible
- The type of both operands is often the same, for example `int` or `float`
- The resulting type is then the type of both operands
- Operands do not modify the current declarations

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{int}, D\rangle \wedge \langle \mathtt{b}, D\rangle \rightarrow \langle \mathtt{int}, D\rangle}{\langle (\mathtt{a}+\mathtt{b}), D\rangle \rightarrow \langle \mathtt{int}, D\rangle}$$

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{float}, D\rangle \wedge \langle \mathtt{b}, D\rangle \rightarrow \langle \mathtt{float}, D\rangle}{\langle (\mathtt{a}+\mathtt{b}), D\rangle \rightarrow \langle \mathtt{float}, D\rangle}$$

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{string}, D\rangle \wedge \langle \mathtt{b}, D\rangle \rightarrow \langle \mathtt{string}, D\rangle}{\langle (\mathtt{a}+\mathtt{b}), D\rangle \rightarrow \langle \mathtt{string}, D\rangle}$$

Typing rules and semantic rules

- The type of both operands could differ, but still be compatible (for example adding an `int` and a `float`)
- The resulting type is then the most generic type of the operands
- Operands do not modify the current declarations

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{int}, D\rangle \wedge \langle \mathtt{b}, D\rangle \rightarrow \langle \mathtt{float}, D\rangle}{\langle (\mathtt{a}+\mathtt{b}), D\rangle \rightarrow \langle \mathtt{float}, D\rangle}$$

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{float}, D\rangle \wedge \langle \mathtt{b}, D\rangle \rightarrow \langle \mathtt{int}, D\rangle}{\langle (\mathtt{a}+\mathtt{b}), D\rangle \rightarrow \langle \mathtt{float}, D\rangle}$$

Typing rules and semantic rules

- Statements in a sequence both modify, top-to-bottom, the declarations
- Usually we expect the statements to simply return nothing, that is `void`
- Further we cannot say anything about what they each do

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{void}, D_1\rangle \wedge \langle \mathtt{b}, D_1\rangle \rightarrow \langle \mathtt{void}, D_2\rangle}{\langle (\mathtt{a};\mathtt{b}), D\rangle \rightarrow \langle \mathtt{void}, D_2\rangle}$$

Typing rules and semantic rules


```
int x = 10;
int y = 20;
x = (x + y);
```

Declarations:

PC
1

Typing rules and semantic rules


```
int x = 10;
int y = 20;
x = (x + y);
```

Declarations:

PC	x
2	int

Typing rules and semantic rules


```
int x = 10;
int y = 20;
x = (x + y);
```

Declarations:

PC	x	y
3	int	int

Typing rules and semantic rules


```
int x = 10;
int y = 20;
x = (x + y);
```

Declarations:

PC	x	y
4	int	int

Typing rules and semantic rules

- The typing rule for an `if-then-else` requires the condition to be a boolean expression, and assumes the type of both the then and the else bodies
- The type of both the then and the else bodies must be the same (usually `void`, something else in case of function returns)
- It does not add anything to the declarations, even though the bodies of the then and the else might declare local variables

$$\frac{\langle \mathtt{c}, D\rangle \rightarrow \langle \mathtt{bool}, D\rangle \wedge \langle \mathtt{A}, D\rangle \rightarrow \langle \mathtt{T}, D'\rangle \wedge \langle \mathtt{B}, D\rangle \rightarrow \langle \mathtt{U}, D'\rangle \wedge T = U}{\langle (\mathtt{if\ c}\ \{\ \mathtt{A}\ \}\mathtt{else}\{\ \mathtt{B}\ \}), D\rangle \rightarrow \langle \mathtt{T}, D\rangle}$$

Typing rules and semantic rules


```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

PC
1

Typing rules and semantic rules


```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

PC	x
2	int

Typing rules and semantic rules

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

PC	x	y
3	int	int

Typing rules and semantic rules

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

x	y		PC
int	int		4

Typing rules and semantic rules

```
1 int x = 10;
2 int y = 20;
3 if((x > y)) {
4   string z = "x";
5   Console.WriteLine(z)
6  } else {
7   string z = "y";
8   Console.WriteLine(z)
9  }
```

Declarations:

x	y		PC	z
int	int		5	string

Typing rules and semantic rules

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

PC	x	y
6	int	int

Typing rules and semantic rules

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

x	y		PC
int	int		7

Typing rules and semantic rules

Type systems

The INFDEV team

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

x	y		PC	z
int	int		8	string

Typing rules and semantic rules

```
int x = 10;
int y = 20;
if((x > y)) {
  string z = "x";
  Console.WriteLine(z)
 } else {
  string z = "y";
  Console.WriteLine(z)
 }
```

Declarations:

PC	x	y
9	int	int

Typing rules and semantic rules

- The typing rule for a `while` loop requires the condition to be a boolean expression, and assumes the type of the body
- The type of the body can be anything (usually `void`, something else in case of function returns)
- It does not add anything to the declarations, even though the body might declare local variables

$$\frac{\langle \texttt{c}, D\rangle \rightarrow \langle \texttt{bool}, D\rangle \wedge \ \langle \texttt{B}, D\rangle \rightarrow \langle \texttt{T}, D_1\rangle}{\langle (\texttt{while c \{ A \}}), D\rangle \rightarrow \langle \texttt{T}, D\rangle}$$

Typing rules and semantic rules

- The typing rule for a `class` declaration adds the class declaration to the declarations with all its fields and methods
- When adding the declaration of the class, we have to check that the types of the method bodies match their declarations
- Assume that `C` is the class name, f_i is the i-th field in the class (of type F_i), and $\mathtt{m}_j$ is the j-th method in the class (with type M_j)

$$\frac{D_1 := D[C \mapsto [..\ f_i \mapsto F_i\ ..\ m_j \mapsto M_j\ ..]] \wedge \langle \mathtt{M_j}\ \mathtt{m_j}, D_1[\mathtt{this} \mapsto C] \rangle \to \langle M_j^1, D_2 \rangle \wedge\ M_j = M_j^1}{\langle (\mathtt{class\ C\ \{\ ..F_i\ f_i..\ \ ..M_j\ m_j..\ \}}), D \rangle \to \langle \mathtt{T}, D_1 \rangle}$$

Typing rules and semantic rules

- When type checking a `method` declaration (within a class declaration we type check its body and compare the result with the type of the declaration
- Assume that `C` is the class name, p_i is the i-th parameter of the method (of type P_i), and `b` is the method body
- The type of a method is of the form $P_1 \times P_2 \times \cdots \times P_n \rightarrow R$, where P_l is the type of the l-th parameter and R is the return type

$$\frac{\langle (\texttt{b}), D[..p_l \mapsto P_l..] \rangle \rightarrow \langle \texttt{R}, D_1 \rangle}{\langle (\texttt{R m(..P}_1\ \texttt{p}_1\texttt{..)b}), D \rangle \rightarrow \langle (P_1 \times P_2 \times \cdots \times P_n \rightarrow R), D \rangle}$$

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC
1

Typing rules and semantic rules

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC	this
4	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter $\times$ int) $\rightarrow$ void

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC	this
5	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter $\times$ int) $\rightarrow$ void

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC	diff	this
7	int	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter
cnt=int
incr=(Counter $\times$ int) $\rightarrow$ void

Typing rules and semantic rules

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC	diff	this
8	int	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter $\times$ int) $\rightarrow$ void

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public void incr(int diff) {
    this.cnt = (this.cnt + diff);
  }
}
```

Declarations:

PC
10

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter $\times$ int) $\rightarrow$ void

Typing rules and semantic rules

- When type checking a `return` statement, we typecheck its argument
- The type of the argument is also the type of the `return` statement
- There is no change to the declarations

$$\frac{\langle \texttt{x}, D\rangle \rightarrow \langle \texttt{T}, D\rangle}{\langle (\texttt{return x}), D\rangle \rightarrow \langle T, D\rangle}$$

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC
1

Typing rules and semantic rules

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC	this
4	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter$\times$int) $\rightarrow$ int

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC	this
5	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter
cnt=int
incr=(Counter $\times$ int) $\rightarrow$ int

Typing rules and semantic rules

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC	diff	this
7	int	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter$\times$int) $\rightarrow$ int

```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC	diff	this
8	int	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter $\times$ int) $\rightarrow$ int

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC	ret	diff	this
9	int	int	Counter

Classes:

Counter
Counter=Counter $\rightarrow$ Counter cnt=int incr=(Counter$\times$int) $\rightarrow$ int

Typing rules and semantic rules


```
class Counter {
  private int cnt;
  public Counter() {
    this.cnt = 0;
  }
  public int incr(int diff) {
    this.cnt = (this.cnt + diff);
    return this.cnt;
  }
}
```

Declarations:

PC
11

Classes:

Counter
Counter=Counter $\rightarrow$ Counter
cnt=int
incr=(Counter $\times$ int) $\rightarrow$ int

- Statements in a sequence might contain `return` statements
- In this case one of them might not return `void`
- Their sequence will assume the non-`void` type

$$\frac{\langle \texttt{a}, D\rangle \rightarrow \langle \texttt{T}, D_1\rangle \wedge \langle \texttt{b}, D_1\rangle \rightarrow \langle \texttt{void}, D_2\rangle}{\langle(\texttt{a};\texttt{b}), D\rangle \rightarrow \langle \texttt{T}, D_2\rangle}$$

$$\frac{\langle \texttt{a}, D\rangle \rightarrow \langle \texttt{void}, D_1\rangle \wedge \langle \texttt{b}, D_1\rangle \rightarrow \langle \texttt{T}, D_2\rangle}{\langle(\texttt{a};\texttt{b}), D\rangle \rightarrow \langle \texttt{T}, D_2\rangle}$$

Typing rules and semantic rules

- Statements in a sequence might contain `return` statements
- They might both return a non-`void` type
- Their sequence will assume the non-`void` type of both, which must be the same

$$\frac{\langle \mathtt{a}, D\rangle \rightarrow \langle \mathtt{T}, D_1\rangle \wedge \ \langle \mathtt{b}, D_1\rangle \rightarrow \langle \mathtt{U}, D_2\rangle \wedge \ T = U}{\langle (\mathtt{a}; \mathtt{b}), D\rangle \rightarrow \langle \mathtt{T}, D_2\rangle}$$

- Sometimes we may look a field f up from an instance x of a class
- This assumes the type of the field, which needs to be looked up in the class descriptor found in the declarations
- No declaration is further modified

$$\frac{\langle \mathtt{x}, D\rangle \rightarrow \langle \mathtt{C}, D\rangle \wedge \langle \mathtt{f}, C\rangle \rightarrow \langle \mathtt{F}, C\rangle \wedge T = U}{\langle (\mathtt{x}.\mathtt{f}), D\rangle \rightarrow \langle \mathtt{F}, D\rangle}$$

Typing rules and semantic rules

- Sometimes we may call a method m up from an instance x of a class and with parameters p_i
- This assumes the return type of the method, provided that all parameter types match the types expected by the method
- No declaration is further modified

$$\frac{\langle \mathtt{x}, D\rangle \rightarrow \langle \mathtt{C}, D\rangle \wedge \ \langle \mathtt{m}, C\rangle \rightarrow \langle (\mathtt{P}_1 \times \mathtt{P}_2 \times \cdots \times \mathtt{P}_n \rightarrow \mathtt{R}), C\rangle \wedge \ \langle \mathtt{p_i}, D\rangle \rightarrow \langle \mathtt{P}'_i, D\rangle \wedge \ \mathtt{P}_i = \mathtt{P}'_i}{\langle (\mathtt{x.f(..p_i..)}), D\rangle \rightarrow \langle \mathtt{R}, D\rangle}$$

Typing rules and semantic rules

- We may call a static method m from class C and with parameters p_i
- This assumes the return type of the method, provided that all parameter types match the types expected by the method
- We do not need to look up the class because it is already specified in the call
- No declaration is further modified

$$\frac{\langle \mathtt{m}, C\rangle \rightarrow \langle (\mathtt{P}_1 \times \mathtt{P}_2 \times \cdots \times \mathtt{P}_n \rightarrow \mathtt{R}), C\rangle \wedge \langle \mathtt{p_i}, D\rangle \rightarrow \langle \mathtt{P}'_i, D\rangle \wedge \mathtt{P}_i = \mathtt{P}'_i}{\langle (\mathtt{C.f}(..\mathtt{p_i}..)), D\rangle \rightarrow \langle \mathtt{R}, D\rangle}$$

Reading typing rules

- The constructor of a class is simply a specially named static method
- It has no further typing rules

Conclusion

Conclusion

Looking back

- Issues with Python
- Static typing as a way to run a coarse simulation of the program

This is it!

The best of luck, and thanks for the attention!