Computer Systems Modeling and Verification (USEEN1)

Part I: Static typing

Some programming languages, such as JavaScript, are dynamically typed: the types are enforced *at run-time*. Other programming languages, such as Java, are statically typed: the types are enforced *before running the program*, either by the IDE static code analysis, or by the compiler.

Python is a dynamically typed language and the default compiler, which is part of CPython[1], does not perform any static type checking. However, type hints[2] are now part of the syntax and a specification of the type system has also been provided[3].

Several static type checkers for Python are currently available:

- Mypy, originally developed by Dropbox, the reference static type checker.
- Pyright, developed by Microsoft, and part of the VS Code extension for Python.
- Pytype, developed by Google.
- Pyre, developed by Facebook and Instagram.
- PyCharm, developed by JetBrains, comes with a built-in type checker.

The main property expected from a static type checker is the following one : "if no *type error* has been detected during static analysis, then no `TypeError` exception should be raised during run-time".

This property is important because that means that you can avoid a lot of unit testing by just running a type checker on your source code. On the other hand, a type checker might reject a correct program (which never raises any `TypeError` exception) if this program does not obey the typing rules of the system.

1 Typing arithmetic expressions

In order to check whether an expression that contains variables is well-typed, we need some environment that tells us what the types of those free variables are. We will represent such a *typing environment* as a mapping from variables to types.

The assertion which states that an expression e is well-typed of type τ under a typing environment Γ, is usually written $\Gamma \vdash e : \tau$ and is called a *typing judgment*.

Now, we need to define which typing judgments are valid and which are not. For that purpose, we rely on a set of *typing rules* to derive the valid typing judgments. Such a set of typing rules is called a *type system*.

Arithmetic expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash i : \texttt{int}}\ (\text{Constant}^{i}) \qquad \frac{\Gamma[\mathrm{x}] = \tau}{\Gamma \vdash \texttt{x} : \tau}\ (\text{Name})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \,\texttt{+}\, e_2 : \texttt{int}}\ (\text{Add}^{ii})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \,\texttt{-}\, e_2 : \texttt{int}}\ (\text{Sub}^{ii}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \,\texttt{*}\, e_2 : \texttt{int}}\ (\text{Mult}^{ii})$$

Note. In rule (Constant^{i}), i is any integer literal.

1. https://devguide.python.org/internals/compiler
2. https://docs.python.org/3/library/typing.html
3. https://typing.readthedocs.io/en/latest

The rule for addition (Add), for instance, can be read as follows:

"if e_1 is a well-typed expression of type `int` under Γ, and e_2 is a well-typed expression of type `int` under Γ, then $e_1 + e_2$ is a well-typed expression of type `int` under Γ."

1.1 Typing derivations

A judgment is then valid if, and only if, one can build a *typing derivation* for this judgment. A derivation is a tree-like structure, where the root is the judgment, the nodes are proper instances of the rules and all the leaves are axioms (that is, rules without hypotheses).

Example

Here is a derivation for judgment $\Gamma \vdash \mathtt{x + 2} : \mathtt{int}$, where $\Gamma = \{\mathtt{x} : \mathtt{int}\}$.

$$\dfrac{\dfrac{\Gamma[\mathtt{x}] = \mathtt{int}}{\Gamma \vdash \mathtt{x} : \mathtt{int}}\,(\text{Name}) \qquad \dfrac{}{\Gamma \vdash \mathtt{2} : \mathtt{int}}\,(\text{Constant}^{i})}{\Gamma \vdash \mathtt{x\ +\ 2} : \mathtt{int}}\,(\text{Add}^{ii})$$

1.2 Implementation

A type system corresponds to a specification: it specifies formally which expression is well-typed of a given type. A type checker is an implementation of such a specification.

The process of type checking takes an expression e and a context Γ and computes a type τ such that $\Gamma \vdash e : \tau$, reporting a type error if such τ does not exist.

For simple type systems, the rules are *syntax-directed*, which means that there is exactly one rule for each syntactic form (and then the rule can be given the name of the AST node).

In this case, the type checker can be directly implemented as a recursive function, defined by case analysis on the AST, that returns the type of the expression (and raises a `TypeError` if such a type does not exist). The recursive calls are thus implicitly building the type derivation.

If we want to compute the type of an expression, we need to define first a datatype for types (even if the only type is `int` for the moment):

```
type Type = Int

@dataclass
class Int: ...
```

Exercises

1. Define a function `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` that returns the type of the expression if it is well-typed (and raises a `TypeError` exception otherwise).

2. Define a function `get_type(e: Expression, m: Mapping[str, Type] = {}) -> str` that returns the type of the expression as a string.

2 Typing integer and string operations

Let us now extend our fragment of Python AST in order to include integer and string operations (with integer and string constants).

String expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash s : \texttt{str}}\ (\text{Constant}^s)$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 + e_2 : \texttt{str}}\ (\text{Add}^{ss})$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 * e_2 : \texttt{str}}\ (\text{Mult}^{si}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 * e_2 : \texttt{str}}\ (\text{Mult}^{is})$$

Note. In rule (Constant^s), s is any string literal.

Example

Here is a derivation for judgment $\Gamma \vdash \texttt{(y + 1) * "a"} : \texttt{str}$, where $\Gamma = \{\texttt{y} : \texttt{int}\}$.

$$\frac{\dfrac{\dfrac{\Gamma[\texttt{y}] = \texttt{int}}{\Gamma \vdash \texttt{y} : \texttt{int}}\ (\text{Name}) \qquad \dfrac{}{\Gamma \vdash \texttt{1} : \texttt{int}}\ (\text{Constant}^i)}{\Gamma \vdash \texttt{y + 1} : \texttt{int}}\ (\text{Add}^{ii}) \qquad \dfrac{}{\Gamma \vdash \texttt{"a"} : \texttt{str}}\ (\text{Constant}^s)}{\Gamma \vdash \texttt{(y + 1) * "a"} : \texttt{str}}\ (\text{Mult}^{is})$$

2.1 Implementation

If we want to extend our type checker for this kind of expressions, we need to first adapt the definition of `Type`.

```
type Type = Int | Str

@dataclass
class Int: ...

@dataclass
class Str: ...
```

Exercises

1. Modify `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` in order to deal with string constants and operations.

2. Modify `get_type(e: Expression, m: Mapping[str, Type] = {}) -> str` in order to deal with string constants and operations.

3 Typing integer, string and boolean operations

Let us now extend our fragment of Python AST in order to include integer, string and boolean operations (with integer, string and boolean constants).

Boolean expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash b : \texttt{bool}}\ (\text{Constant}^{b})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{int}}\ (\text{Add}^{bb})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{int}}\ (\text{Add}^{bi}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{int}}\ (\text{Add}^{ib})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{-} e_2 : \texttt{int}}\ (\text{Sub}^{bb})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{-} e_2 : \texttt{int}}\ (\text{Sub}^{bi}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{-} e_2 : \texttt{int}}\ (\text{Sub}^{ib})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{int}}\ (\text{Mult}^{bb})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{int}}\ (\text{Mult}^{bi}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{int}}\ (\text{Mult}^{ib})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{str}}\ (\text{Mult}^{bs}) \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{str}}\ (\text{Mult}^{sb})$$

Note. In rule (Constant^{b}), b is any boolean literal (that is, either `True` or `False`).

Example

Here is a derivation for judgment $\Gamma \vdash$ `(y + "a") * z` : `str`, where $\Gamma = \{$`y` : `str`, `z` : `bool`$\}$.

$$\frac{\dfrac{\dfrac{\Gamma[\texttt{y}] = \texttt{str}}{\Gamma \vdash \texttt{y} : \texttt{str}}\ (\text{Name}) \qquad \dfrac{}{\Gamma \vdash \texttt{"a"} : \texttt{str}}\ (\text{Constant}^{s})}{\Gamma \vdash \texttt{y + "a"} : \texttt{str}}\ (\text{Add}^{ss}) \qquad \dfrac{\Gamma[\texttt{z}] = \texttt{bool}}{\Gamma \vdash \texttt{z} : \texttt{bool}}\ (\text{Name})}{\Gamma \vdash \texttt{(y + "a") * z} : \texttt{str}}\ (\text{Mult}^{sb})$$

3.1 Implementation

If we want to extend our type checker for this kind of expressions, we need to first adapt the definition of `Type`.

```
type Type = Int | Str | Bool

@dataclass
class Int: ...

@dataclass
class Str: ...

@dataclass
class Bool: ...
```

Exercises

1. Modify `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` in order to deal with boolean constants and operations.

2. Modify `get_type(e: Expression, m: Mapping[str, Type] = {}) -> str` in order to deal with boolean constants and operations.

3.2 Adding comparison operators

Let us now extend our fragment of Python AST in order to include comparison operators. The The typing rules for operators `==`, `!=` and `>` are given below. The rules for the remaining comparison operators (`<`, `>=` and `<=`) are similar to `>`.

Comparison operators $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \texttt{==} e_2 : \texttt{bool}}\ (\text{Eq}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \texttt{!=} e_2 : \texttt{bool}}\ (\text{NotEq})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{>} e_2 : \texttt{bool}}\ (\text{Gt}^{ii}) \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{>} e_2 : \texttt{bool}}\ (\text{Gt}^{bb})$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{>} e_2 : \texttt{bool}}\ (\text{Gt}^{bi}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{>} e_2 : \texttt{bool}}\ (\text{Gt}^{ib})$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \texttt{>} e_2 : \texttt{bool}}\ (\text{Gt}^{ss})$$

Note. In rules (Eq) and (NotEq), τ_1 and τ_2 are arbitrary types.

Example

Here is a derivation for judgment $\Gamma \vdash \texttt{(y == "a") > z} : \texttt{bool}$, where $\Gamma = \{\texttt{y} : \texttt{int}, \texttt{z} : \texttt{int}\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma[\texttt{y}] = \texttt{int}}{\Gamma \vdash \texttt{y} : \texttt{int}}\ (\text{Name}) \qquad \dfrac{}{\Gamma \vdash \texttt{"a"} : \texttt{str}}\ (\text{Constant}^{s})}{\Gamma \vdash \texttt{y == "a"} : \texttt{bool}}\ (\text{Eq}) \qquad \dfrac{\Gamma[\texttt{z}] = \texttt{int}}{\Gamma \vdash \texttt{z} : \texttt{int}}\ (\text{Name})}{\Gamma \vdash \texttt{(y == "a") > z} : \texttt{bool}}\ (\text{Gt}^{bi})$$

Exercises

1. Modify `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` in order to deal with comparison operators.

2. (*optional*). Add the cases for the remaining comparison operators (`<`, `>=` and `<=`).

3.3 Adding logical operators

Let us now extend our fragment of Python AST in order to include binary logical operators. The typing rules for logical operator `and` (conjunction) are given below. The rules for logical operator `or` (disjunction) are similar to `and`.

In Python, these logical operators can be applied to any types which are convertible to bool. We will provide typing rules only for the cases where both arguments have the same type. Some correct Python expressions are thus rejected by our type system.

Boolean operators $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \,\texttt{and}\, e_2 : \texttt{bool}}\ (\text{And}^{bb})$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \,\texttt{and}\, e_2 : \texttt{int}}\ (\text{And}^{ii})$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 \,\texttt{and}\, e_2 : \texttt{str}}\ (\text{And}^{ss})$$

Example

Here is a derivation for judgment $\Gamma \vdash \texttt{(y != "a") and z} : \texttt{bool}$, where $\Gamma = \{\texttt{y} : \texttt{int}, \texttt{z} : \texttt{bool}\}$.

$$\dfrac{\dfrac{\dfrac{\Gamma[\texttt{y}] = \texttt{int}}{\Gamma \vdash \texttt{y} : \texttt{int}}\ (\text{Name}) \qquad \dfrac{}{\Gamma \vdash \texttt{"a"} : \texttt{str}}\ (\text{Constant}^{s})}{\Gamma \vdash \texttt{y != "a"} : \texttt{bool}}\ (\text{Eq}) \qquad \dfrac{\Gamma[\texttt{z}] = \texttt{bool}}{\Gamma \vdash \texttt{z} : \texttt{bool}}\ (\text{Name})}{\Gamma \vdash \texttt{(y != "a") and z} : \texttt{bool}}\ (\text{And}^{bb})$$

Exercises

1. Modify `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` in order to deal with logical operators.

2. (*optional*). Add the cases for the remaining binary logical operator (`or`).

4 Adding user-defined functions

Let us consider simple user-defined functions, with explicit type annotations, such as the following one:

```
def f(x: int, y: int) -> int:
    return x + y
```

The general syntax of such a definition is thus:

```
def f(x1: τ1, ..., xn: τn) -> τ:
    return e
```

Given a typing environment Γ, type checking such a function definition simply amounts to type checking the following judgment: $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$.

Example

Here is a derivation for judgment which states that function `f` is well-typed: $\Gamma \vdash$ `x + y` : `int`, where $\Gamma = \{$`x` : `int`, `y` : `int`$\}$.

$$\dfrac{\dfrac{\Gamma[\texttt{x}] = \texttt{int}}{\Gamma \vdash \texttt{x} : \texttt{int}}\,(\text{Name}) \qquad \dfrac{\Gamma[\texttt{y}] = \texttt{int}}{\Gamma \vdash \texttt{y} : \texttt{int}}\,(\text{Name})}{\Gamma \vdash \texttt{x + y} : \texttt{int}}\,(\text{Add}^{ii})$$

Let us now focus on function calls. The types of the parameters are not available, and the types of the actual arguments can thus be statically checked. Here is the rule required to type check function calls:

Function call $\boxed{\Gamma \vdash e : \tau}$

$$\dfrac{\Gamma \vdash f : (\tau_1, \ldots, \tau_n) \rightarrow \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}\,(\text{Call})$$

Example

Here is a derivation for judgment $\Gamma \vdash$ `f(x, 2)` : `int`, where `f` is the function given above (which is well-typed) and $\Gamma = \{$`x` : `int`, `f` : `(int, int) -> int`$\}$.

$$\dfrac{\dfrac{\Gamma[\texttt{f}] = \texttt{(int, int) -> int}}{\Gamma \vdash \texttt{f} : \texttt{(int, int) -> int}}\,(\text{Name}) \qquad \dfrac{\Gamma[\texttt{x}] = \texttt{int}}{\Gamma \vdash \texttt{x} : \texttt{int}}\,(\text{Name}) \qquad \dfrac{}{\Gamma \vdash \texttt{2} : \texttt{int}}\,(\text{Constant}^{i})}{\Gamma \vdash \texttt{f(x, 2)} : \texttt{int}}\,(\text{Call})$$

If we want to extend our type checker for this kind of expressions, we need to first adapt the definition of `Type` in order to represent function types such as $(\tau_1, \ldots, \tau_n) \rightarrow \tau$.

```
type Type = Int | Str | Bool | Fun

@dataclass
class Fun:
    params: list[Type]
    result: Type

# ...
```

Exercises

1. Modify `get_expr_type(e: expr, m: Mapping[str, Type] = {}) -> Type` in order to deal with function calls.