University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2025

Exercise 4: Types

Strict deadline for uploading solutions: **June 18, 2025, 11:59pm**, Stuttgart time

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with the folder structure and the templates that must be used for the submission:

```
exercise4.zip
├─ task_1
│  └─ task1.csv (Edit/replace this file with your solution)
├─ task_2
│  ├─ task2.py (Edit/replace this file with your solution)
│  └─ task2.csv (Edit/replace this file with your solution)
└─ task_3
   ├─ .devcontainer
   │  └─ devcontainer.json
   └─ exercise4
      ├─ src
      │  ├─ expression.rs
      │  ├─ type_checker.rs (Edit/replace this file with your solution)
      │  └─ lib.rs
      ├─ tests
      │  └─ type_checker_test.rs
      ├─ Cargo.toml
      └─ Cargo.lock
```

Figure 1: Directory structure in the provided .zip file and in your uploaded solution file.

The submission must be compressed in a .zip file (not .rar, .7z, .g.z, or any other format) using exactly the folder structure and names shown above.

1 Task I (20% of total points of the exercise)

This task is to check your understanding of some concepts around types. For that, we give you code snippets written in different made-up programming languages. We describe relevant parts of the static semantics (e.g., errors during compilation) and dynamic semantics (i.e., runtime behavior) of each language with comments inside the snippet. You have to **answer one question per snippet** by selecting **exactly one** correct answer out of multiple choices.
To submit your answer, please fill in the file *exercise4/task_1/task1.csv*. One row corresponds to one code snippet, which corresponds to one answer. Fill in **A** in the second column of the CSV file to select option A as the answer, **B** to select option B, etc.

Code Snippet 1

```
indices = [0, 1, 2]
values = [10, 20, 30]
for i in indices:
  print(values[i/2]) # At runtime:
  # "TypeError: list indices must be integers or slices, not float"
```

What property of the language's type system can you deduct from Code Snippet 1?

A This is an example of a dynamically typed language.

B This is an example of a statically typed language.

C This is an example of a language without types.

Code Snippet 2

```
int x = 10;
String y = "32";
int z = x + y; // At compilation time:
// "Error: incompatible types: int cannot be converted to String"
```

What property of the language's type system can you deduct from Code Snippet 2?

A This is an example of a dynamically typed language.

B This is an example of a statically typed language.

C This is an example of a language without types.

Code Snippet 3

```
def find_in_list(lst, value):
  target_index = "Not found"
  for i in range(len(lst)):
    if lst[i] == value:
      target_index = i
      break
  return target_index

result = find_in_list([1, 2, 3], 2)
```

Which statement is true for Code Snippet 3?

A This is an example of Polymorphic variables.

B This is an example of Parametric polymorphism.

C This is an example of Subtype polymorphism.

Which of the following terms in lambda calculus evaluates to 0?

A $(\lambda x \,.\, \lambda y \,.\, x)\ 0\ (succ\ 0)$

B $(\lambda x \,.\, \lambda y \,.\, y)\ 0\ (succ\ 0)$

C $(\lambda x \,.\, \lambda y \,.\, x\ y)\ (succ\ 0)\ 0$

D $(\lambda x \,.\, \lambda y \,.\, y\ x)\ (succ\ 0)\ 0$

2 Task II (30% of total points of the exercise)

This task is about **writing type annotations** in Python and identifying subtle type-related bugs. Recent versions of Python support optional type annotations for functions (since Python 3.5[1]) and local variables (since Python 3.6[2]). Those can be checked by a third-party program (e.g., `mypy`) before running the program. You can find a quick overview of the format of the annotations and the possible types here: `https://mypy.readthedocs.io/en/latest/cheat_sheet_py3.html`.

2.1 Add Type Annotations

Your task is to extend a Python 3 program by adding type annotations. You should add the most specific types possible such that the program is still well-typed. E.g., the function `def add(a, b): return a + b` should be annotated with types `def add(a: int, b: int) -> int`. Please annotate all the methods (both arguments and return types) in the file *exercise4/task_2/task2.py*.
Local variables do not need to be annotated. Note that types can also be generic, e.g., `list[str]` is a valid type. For such generic types, please specify all type arguments, e.g., `dict[int, int]` not just `dict`.
Note: You should use the new type annotation syntax introduced in Python 3.10, i.e., `list[str]` instead of `List[str]`. You are not allowed to use the old syntax, i.e. the types from Typing module.
Below is an example of a Python program with type annotations:

Example of Annotated Python Functions

```python
def add(a: int, b: int) -> int:  # Annotate function args and return type
  result = a + b  # No need to annotate local variables
  return result

def greet(name: str) -> str:  # Annotate here
  greeting = f"Hello, {name}!"  # No need to annotate local variables
  return greeting

def divide(x: float, y: float) -> float:  # Annotate here
  if y == 0:
    raise ValueError("Cannot divide by zero")
  quotient = x / y  # No need to annotate local variables
  return quotient
```

2.2 Identify Type-Related Bugs

Additionally, you need to find and identify any subtle type-related bugs in the provided code (Hint: there are three bugs). Write your answer in the file *exercise4/task_2/task2.csv*. You must provide the line number of the statements with type error, mention it with respect to the original file.

Evaluation Criteria:

To help yourself in the identification of bugs, you might want to use the `mypy` tool to check the type annotations. For evaluation, every provided type annotation is compared with the correct, most specific one for that argument/return value.

Useful Commands:

Run Mypy with this command (from the folder containing *task2.py*)

```
docker run --rm -v $(pwd):/data cytopia/mypy task2.py
```

To execute your Python file, use the command:

```
docker run --rm -v $(pwd):/home python:3.13-slim python /home/task2.py
```

Note that docker commands are provided for your convenience, however you can still install `mypy` and `python` on your local machine and run the commands without docker.

[1] `https://www.python.org/dev/peps/pep-0484/`
[2] `https://www.python.org/dev/peps/pep-0526/`

3 Task III (50% of total points of the exercise)

This task is about implementing a very **simple type checker** based on the formal type systems introduced in the lecture. That is, given a grammar of a language, a set of type rules, and an expression in the language, the type checker should perform a typing derivation of the expression until either all the hypotheses of all type rules are fulfilled (i.e., the expression is well-typed) or no type rules can be applied (i.e., the expression is not well-typed).

Figure 2 specifies the language for this task and its type system. Each expression in the language has one out of three types: Num (for integer numbers), Bool (for booleans), and None (for the None type).

$$
\begin{array}{lll}
e ::= & \texttt{None} & \text{None} \\
 & \mid \texttt{true} \mid \texttt{false} & \text{boolean literals} \\
 & \mid 0 \mid 1 \mid 2 \mid \ldots \mid n & \text{integer literals} \\
 & \mid -e & \text{unary minus} \\
 & \mid e + e & \text{integer addition} \\
 & \mid e \mathbin{||} e & \text{boolean disjunction} \\
 & \mid e = e & \text{equality} \\
 & \mid e \;\texttt{if}\; e \;\texttt{else}\; e & \text{ternary if-else} \\
 & \mid e \# e \# e & \text{ternary equality}
\end{array}
$$

(a) Expression grammar. The intuition for each construct is given in gray on the right.

$$
\frac{}{\texttt{true} : \mathit{Bool}}\ \text{T-True}
\qquad
\frac{}{\texttt{false} : \mathit{Bool}}\ \text{T-False}
$$

$$
\frac{}{\texttt{None} : \mathit{None}}\ \text{T-None}
\qquad
\frac{}{n : \mathit{Num}}\ \text{T-Num}
$$

$$
\frac{e : \mathit{Num}}{-e : \mathit{Num}}\ \text{T-Neg}
\qquad
\frac{e_1 : \mathit{Num} \quad e_2 : \mathit{Num}}{e_1 + e_2 : \mathit{Num}}\ \text{T-Add}
$$

$$
\frac{e_1 : \mathit{Bool} \quad e_2 : \mathit{Bool}}{e_1 \mathbin{||} e_2 : \mathit{Bool}}\ \text{T-Or}
\qquad
\frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : \mathit{Bool}}\ \text{T-Eq}
$$

$$
\frac{e_1 : T \quad e_2 : \mathit{Bool} \quad e_3 : T}{e_1 \;\texttt{if}\; e_2 \;\texttt{else}\; e_3 : T}\ \text{T-If}
$$

$$
\frac{e_1 : T \quad e_2 : T \quad e_3 : T}{e_1 \# e_2 \# e_3 : \mathit{Bool}}\ \text{T-TEq1}
\qquad
\frac{e_1 : T_1 \quad e_2 : T_1 \quad e_3 : T_2}{e_1 \# e_2 \# e_3 : \mathit{Bool}}\ \text{T-TEq2}
$$

$$
\frac{e_1 : T_2 \quad e_2 : T_1 \quad e_3 : T_1}{e_1 \# e_2 \# e_3 : \mathit{Bool}}\ \text{T-TEq3}
\qquad
\frac{e_1 : T_1 \quad e_2 : T_2 \quad e_3 : T_1}{e_1 \# e_2 \# e_3 : \mathit{Bool}}\ \text{T-TEq4}
$$

(b) Type rules. The hypotheses are above the line, the conclusion is below the line, the rule name to the right. Type rules without hypotheses are axioms.

Figure 2: Grammar and type rules for a simple language with boolean and arithmetic expressions.

Note: The ternary equality operator `#` is a special operator that checks whether any two of its three operands are equal. Before implementing the type checker for this operator, read the type rules for it (T-TEq1 – T-TEq4) carefully.

Before starting to implement an automated type checker, we recommend writing down (with pen and paper) the typing derivations for a few expressions in the given language.

A template for your implementation is given in *exercise4/task_3/*. You will need to implement the `visit` methods in the `TypeChecker` struct, in the *type_checker.rs* file. Each `visit` method therein takes an expression as argument and returns its type if type checking is successful, or returns a `TypeError` if type checking fails. The `TypeError` struct takes two types as arguments, which can be two incompatible types or the expected vs. the actual type of an expression.

There is no need to parse the input expression because you are already given its AST. For the AST definition, see *expression.rs*. For debugging, you can print expressions via their `to_string` method. (It always adds parentheses, to make the infix operators unambiguous.)

To test your implementation, use the tests in *type_checker_test.rs* as a starting point. We strongly recommend implementing additional tests, e.g., ill-typed expressions and more complex expressions.

For more details on handling nested expressions and recursive visiting, see Appendix A.1.

Environment Setup: For this assignment you will need:

- A compatible Rust compiler (e.g., rustc 1.83.0 or later, which will be used for grading)
- Cargo

- We provide a convenient way to run your code in a Docker container. To use it, you need to have Docker installed on your machine and follow this guide: Quick Start: Open an existing folder in a container. Use the configuration in the folder: *exercise4/task_3/.devcontainer/devcontainer.json*.

Evaluation Criteria: Your solution will be built with Cargo and then be tested against a set of type-correct and type-incorrect expressions. Solutions that cannot be built with Cargo cannot be graded. Your solution must not modify any code besides in the *type_checker.rs* file. In case you implement additional tests in *type_checker_test.rs*, which we strongly recommend, do not send them. *type_checker_test.rs* should be submitted exactly as received.

Useful Commands: To build and run the tests, execute the following:

- `cargo build`
- `cargo test`

A Appendix

The following provides additional information on recursive data structures and how to visit them, which is useful for implementing the type checker in Task III.

A.1 Recursive Data Structures

A **recursive data structure** is a data structure that is defined in terms of smaller instances of the same type of structure. In simpler terms, a part of the structure can contain another instance of itself. This property allows recursive data structures to represent hierarchical or nested information efficiently.

Common Examples

1. A **list** where each element points to the next element (e.g., linked lists).
2. A **tree** where each node has child nodes (e.g., binary trees, expression trees).

Recursive Definition: Example

Consider a tree-like structure used to represent mathematical expressions:
Expression (AST):

- A literal (e.g., a number like `3` or a boolean like `true`).
- A binary operation combining two expressions (e.g., `e1 + e2`).
- An if-then-else statement combining three expressions (e.g., `if e1 then e2 else e3`).

This definition is recursive because an expression contains sub-expressions.

Visiting Recursive Data Structures

To **visit** a recursive data structure means to systematically explore all parts of it. This is often done using **traversal algorithms**. In trees, for example, common traversal strategies are *pre-order*, *in-order*, and *post-order*.
Example: Pre-order Traversal

- Visit the current node.
- Recursively visit the left subtree.
- Recursively visit the right subtree.

Python Example: Expression Visitor

```python
def visit(node):
    if isinstance(node, Literal):
        print(f"Literal: {node.value}")
    elif isinstance(node, BinaryOp):
        print(f"Operator: {node.operator}")
        visit(node.left)
        visit(node.right)
    elif isinstance(node, IfElse):
        print("If statement")
        visit(node.condition)
        visit(node.then_branch)
        visit(node.else_branch)
```

Effective Metaphor

Imagine a **nesting doll** (matryoshka):

- Each doll contains a smaller doll inside it.
- To fully explore the nesting doll, you must *open* each doll and examine what it contains.

- Similarly, in a recursive data structure, you *open* each node and explore its components.

By understanding and implementing these ideas, you will be prepared to work with recursive data structures like Abstract Syntax Trees (ASTs).