University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2020

Exercise 5: Types

(Deadline for uploading solutions: July 4, 2020, 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 directory structure and the templates that must be used for the submission**.

The directory structure is:

```
exercise5/
├─ task1/
│   ├─ reference-counting.csv
│   └─ mark-and-sweep.csv
├─ task2/
│   ├─ expression1.json
│   ├─ expression2.json
│   ├─ expression3.json
│   ├─ expression4.json
│   └─ expression5.json
├─ task3.csv
├─ task4/
│   ├─ structA.csv
│   ├─ structB.csv
│   └─ structC.csv
└─ task5.py
```

The submission must be compressed in a zip file using the given directory structure. The name of files and directories must not be changed or moved, otherwise the homework will not be evaluated.

General tips for making sure your submission is graded as you expect:

- Use only the zip format for the archive (**not rar**, 7z, etc.).
- Your zip file should contain exactly one top-level directory `exercise5/`, as in the zip file provided by us.
- Do not rename files or folders, simply open the files provided and put your solutions in them.

1 Task I (15% of total points of the exercise)

This task is about **garbage collection**, or **automatic memory management**, in "managed" languages like Java, Python, Objective-C, and many others. We give you an excerpt of a program written in Java that allocates several objects on the heap and performs several assignments. You have to determine at certain statements in the program, how two different types of automatic memory management would keep track of the objects. Objects are identified by their `value` field. For example, "object `A`" refers to the object created via `new Object("A")`.

- For **reference counting**, you shall give the current reference count as an integer number for all objects in the program. The reference count of an object is the number of pointers referring to that object. For example, at line 11 below, you would put `2` as the reference count of object `A`.
- For **mark-and-sweep garbage collection**, you shall determine for each object whether it is marked, that is, whether it is reachable from the current variable definitions. That is, assume a mark-and-sweep garbage collector (GC) is invoked at lines 11, 15, 18, 23, and 28 of the program below.[1] Enter `true` if the object is marked (and thus won't be deallocated), and `false` if not. For example, at line 11 below, you would put `true` for object `A`.

For objects that have not yet been created at a certain point in the program, submit `undefined` as the reference count and for the marked property. For example, object `B` is not yet created at line 11 and thus has `undefined` as the reference count and marked status in the respective answer files.

The Java program in the following:

```java
1  class Object { // Simple class with data and pointer to another instance.
2      String value;
3      Object next = null;
4      // Constructor
5      Object(String value) { this.value = value; }
6  };
7
8  // In a method body somewhere else in the program...
9  Object o1 = new Object("A");
10 Object o2 = o1;
11 // Reference count of object A is 2. Mark-and-sweep GC has marked object A.
12
13 Object o3 = new Object("B");
14 o3.next = o2;
15 // Submit answer in CSV.
16
17 o2 = null;
18 // Submit answer in CSV.
19
20 Object o4 = new Object("C");
21 o4.next = o3;
22 o1.next = o4;
23 // Submit answer in CSV.
24
25 o1 = null;
26 o3 = null;
27 o4 = null;
28 // Submit answer in CSV.
```

Fill your answers in the files *exercise5/task1/reference-counting.csv* and *exercise5/task1/mark-and-sweep.csv*. The first column gives the line number at which point you shall give the current state of the heap-allocated objects. The remaining columns correspond to the objects in the program.
Please make sure that your CSV file is valid. For example, it should not contain additional commas or tabs or semicolons as delimiters. Try to stay away from Excel or LibreOffice, or double check your files with a plain-text editor after editing them in said programs.

Evaluation Criteria: Your solution will be compared against the correct result for each object and each of the two memory management systems.

[1]In the real-world, the GC is often invoked non-deterministically, which is one of the drawbacks of GCs.

2 Task II (25% of total points of the exercise)

This task is about **manual type checking** as it was shown in the lecture. That is, given a grammar of a language, a set of type rules, and an expression in the language, perform a typing derivation of the expression until either all type rules' hypotheses are fulfilled (and the expression is well-typed) or no more type-rules can be applied (and the expression is not well-typed).

Figure 1 specifies the language and its type system for this task. To make parsing complex expressions unambiguous, we add parantheses where needed (which do not carry any additional meaning besides clarifying the order of operations). A type of T in a type rule can be instantiated with any concrete type (e.g., Nat or $Bool$), but it must be consistent across one application of the type rule.

$e ::=$ `true` | `false`	boolean literals
| n	integer literals, so $n \in \mathbb{N}$
| $!e$	boolean negation
| $e \mathbin{\&\&} e$	boolean conjunction
| $e + e$	integer addition
| $e = e$	equality
| `if` e `then` e `else` e	if-then-else

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

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

$$\frac{e_1 : \mathit{Bool} \quad e_2 : \mathit{Bool}}{e_1 \mathbin{\&\&} e_2 : \mathit{Bool}}\ \text{T-And} \qquad \frac{e_1 : \mathit{Nat} \quad e_2 : \mathit{Nat}}{e_1 + e_2 : \mathit{Nat}}\ \text{T-Add}$$

$$\frac{e : \mathit{Bool}}{!e : \mathit{Bool}}\ \text{T-Not} \qquad \frac{e_1 : T \quad e_2 : T}{e_1 = e_2 : \mathit{Bool}}\ \text{T-Eq}$$

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

(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 1: Grammar and type rules for a simple language with boolean and arithmetic expressions.

It makes sense to write down the typing derivations first with pen and paper. But your final answers must be encoded into JSON for the submission. The encoding is similar to how you encoded ASTs to JSON in the first exercise. Each application of a type rule in the typing derivation tree is encoded into a JSON object with the following properties:

- `"expression"`: the expression that is type checked as a string, e.g., `"if true then 1 else 2"`,
- `"type"`: the type of the current expression as a string, e.g., `"Nat"` for the previous expression,
- `"rule"`: the name of the applied type rule (see Figure 1 (b)) as a string, e.g., `"T-If"`, and
- `"hypotheses"`: an array of resursively JSON-encoded type rules, one for each of the hypotheses of the applied rule.

As an example of the JSON encoding, we encode the following typing derivation

$$\dfrac{\dfrac{}{\texttt{false} : \mathit{Bool}}\ \text{T-False} \qquad \dfrac{}{\texttt{true} : \mathit{Bool}}\ \text{T-True}}{\texttt{false} \mathbin{\&\&} \texttt{true} : \mathit{Bool}}\ \text{T-And}$$

into this JSON (going bottom-up in the derivation tree and recursively encoding each hypothesis from left-to-right until we reach an axiom, i.e., a rule without hypotheses):

```
{
  "expression": "false␣&&␣true",
  "type": "Bool",
  "rule": "T-And",
  "hypotheses": [
    {
      "expression": "false",
      "type": "Bool",
      "rule": "T-False",
      "hypotheses": []
    },
    {
      "expression": "true",
      "type": "Bool",
      "rule": "T-True",
      "hypotheses": []
    }
  ]
}
```

To submit your answer, please fill each of the *expression*.json* files in the *exercise5/task2/* directory with the JSON encoding of your typing derivation for each of the five expressions given below. Whitespace does not matter in the `expression` property value. In all other keys and values of the JSON, whitespace should not occur. All strings are case-sensitive. That is, `T - True` or `t-true` are incorrect rule names.

When you reach a hypothesis to which no typing rule can be applied, use the special string `"invalid"` in the `"rule"` property to mark that your typing derivation ends here (and the original expression is thus not well-typed). E.g., the following typing derivation of the expression !2 ends because 2 is not a $Bool$.

$$\dfrac{\dfrac{}{2\ :Bool}\ \text{invalid}}{!2\ :Bool}\ \text{T-Not}$$

It would be encoded in JSON as:

```
{
  "expression": "!2",
  "type": "Bool",
  "rule": "T-Not",
  "hypotheses": [ {
      "expression": "2",
      "type": "Bool",
      "rule": "invalid", // Note the rule name "invalid".
      "hypotheses": [] // No further hypotheses, since no type rule applies.
  } ]
}
```

Please perform the typing derivations and submit them as JSON for the following expressions:

1. `1 + 2`
2. `true + 2`
3. `(true = true) && (2 = 1)`
4. `if (2 + 2) = 4 then false else true`
5. `if !true then 2 else false`

To make sure that your answers are at least correct JSON syntax, you can use a JSON linting tool (like JSONlint[2] or a text editor with JSON syntax highlighting, e.g., Sublime text, VS Code, Notepad++, IntelliJ IDEA or many others. Also avoid copy-pasting JSON from this PDF, since it often results in superfluous whitespace or wrong characters.

Evaluation Criteria: Your solution will be compared against the correct solutions (the full, correct typing derivation for well-typed expressions, multiple possible partial typing derivations for non well-typed expressions). Whitespace and parentheses are ignored in the expressions.

[2] `https://jsonlint.com/`

3 Task III (20% of total points of the exercise)

This task is about **pointer arithmetic and arrays** in C and C++. You are provided with an incomplete C program and possible code fragments to be used to complete the code. Please fill in some of the blanks with some of the provided code fragments. You can fill in at most one code fragment per blank. Each fragment can be used only once. When completed, the `main` function of the program should be syntactically correct and type-correct, and return the result 13.

Submit the correct assignments of blanks to code fragments in the solution file *exercise5/task3.csv*. In the provided template file, for each blank either fill in the number of the correct fragment or 0 to indicate that none of the fragments should be inserted.

Incomplete Code

```c
#include<stdlib.h>
int main() {
  __blank1__
  char array2[] = {1, 2, 3, 4, 5};
  for (int i = 0; i < 16; i++) {
    array1[i] = i*i;
    __blank2__
  }
  __blank3__
  int sum = 0;
  __blank4__
  free(array1);
  __blank5__
  __blank6__
  return sum;
}
```

Options for code fragments to insert:

- Fragment 1: `int array1[] = {0, 0, 0, 0};`
- Fragment 2: `sum += *(array1 + 3 * sizeof(int));`
- Fragment 3: `sum += array2;`
- Fragment 4: `sum += *(array1 + 3);`
- Fragment 5: `sum += array1 + 3;`
- Fragment 6: `int * array1 = malloc(16 * sizeof(int));`
- Fragment 7: `int * array1 = malloc(16);`
- Fragment 8: `sum += array2[2];`
- Fragment 9: `sum += array2[3];`

Evaluation Criteria: Your solution will be compared against a correct assignment of blanks to code fragments.

4 Task IV (20% of total points of the exercise)

This task is about the **memory layout** of structs and **alignment** in C (or other "systems" languages). Given several definitions of structs in C and several sets of rules, you should compute the memory layout of each struct for each set of rules. That is, you should determine at which offset each field starts and the overall size of the struct in memory.

For the sizes of primitive types, assume 1 byte for `char`s, 4 bytes for `int`s and `float`s, and 8 bytes for pointers. For the natural alignment of primitive types (e.g., where memory access of the underlying architecture would be fastest), assume no alignment requirement for `char`s, 4 byte alignment for `int`s, and 8 byte alignment for `float`s and pointers. For example, an alignment requirement of 4 bytes means that the byte offset of that field must divide without remainder by 4. The first offset in each struct is 0. Similar to C[3], assume no padding between array elements, but there can be before the array to satisfy the alignment of the first element.

You have to compute the *most compact* memory layout for each struct under the following rules:

- **Packed**: The natural alignment requirements (see above) do not have to be respected, i.e., each field can start at an arbitrary offset. The order of fields must not be changed.
- **Default**: The above alignment requirements must be respected for each field. The order of fields must not be changed.
- **Reordered**: The above alignment requirements must be respected for each field, but the order of fields can be changed compared with the declaration. (But not across structs, that would change semantics.)

The three struct definitions are:

Struct A

```c
struct A {
  char  field1;
  float field2;
  int   field3;
};
```

Struct B

```c
struct B {
  char   field1[10];
  int    field2;
  float *field3;
};
```

Struct C

```c
struct C {
  int    *field1;
  struct {
    int   field2;
    float field3;
  } nested_struct;
};
```

For solving the task, it may be useful to draw layout figures with pen and paper, similar to the lecture. However, your final answers must be submitted in the *struct*.csv* files in the *exercise5/task4/* directory. There is one row for each field of the struct and a final row for the overall size in bytes. For the fields, fill in the offset (i.e., the byte index at which the field begins), viewed from the start of the outermost struct, in the columns corresponding to each of the three rules.

To give an example: Under the **Packed** rule, `struct A` is layed out as follows. `field1` starts at byte offset 0, followed by `field2` at byte offset 1, followed by `field3` at byte offset 5. The overall size of the struct is 9 bytes. This solution is already filled into the second column of the solution template in *exercise5/task4/structA.csv*.

Evaluation Criteria: Your solution will be compared against the correct byte offsets of each struct field under each set of alignment requirements.

[3] see https://stackoverflow.com/questions/1066681/can-c-arrays-contain-padding-in-between-elements

5 Task V (20% of total points of the exercise)

This task is about **writing type annotations** in Python. Recent versions of Python support optional type annotations for functions (since Python 3.5[4]) and local variables (since Python 3.6[5]). 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`.

Your task is to extend a Python 3 program that is only partially type-annotated with more type annotations. You should add the most specific types possible such that the program is still well-typed. E.g., the assignment `some_variable = 3` should be annotated with type `int` and not with `Any` (since the latter is always trivially type-correct but not very useful). That is, the correct solution would be `some_variable: int = 3`. Please annotate all the functions (both arguments and return type) and local variables in the file *exercise5/task5.py*. The given file is a type-correct, but only partially type-annotated Python 3.6 program. Loop variables (e.g., `i` in `for i in` ...) and variables in list comprehensions 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`.

Evaluation Criteria: Your code will be evaluated by type-checking it with `mypy` version 0.780 to ensure that it is well-typed. Additionally, every provided type annotation is compared with the correct, most specific one for that variable/argument/return value.

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