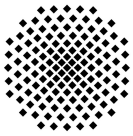

Universität Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart, Winter 2019/2020

Exercise 5: Types

(Deadline for uploading solutions: January 12, 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.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 (**not rar** or other formats) using the given directory structure. The names of directories and files must not be changed, otherwise the homework will not be evaluated. Late submissions will be not accepted.

There are five tasks, which contribute a specific percentage to the overall points for this exercise.

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

This task is about the difference between **nominal and structural type systems**. We give you several code-snippets written in a simple "toy language". You have to determine if the **last statement** of each code snippet is well-typed, if we assume (A) a nominal type system or (B) a structural type system.

Our toy language is inspired by JavaScript with types, similar to Flow or TypeScript.[1] For example, given the following code in our toy language:

```
type Foo = {
  field: String
};
let foo: Foo = { field : "somestring" };
let bar: Foo = foo; // Is this line well-typed (=valid) or not (=invalid)?
```

[1] https://flow.org/en/docs/getting-started/, https://www.typescriptlang.org/docs/handbook/basic-types.html

Lines 1 to 3 declare a new record type (or "object type" in JavaScript) with name `Foo` that has a single field (or property) with name `field` of type `String`. Line 4 declares a variable of name `foo` with type `Foo` and initializes it with a type-compatible object literal. Now, the question is whether the final line 5 is well-typed.
For type assignments, such as:

```
1 type TypeB = TypeA;
```

we assume that `TypeB` is a new, distinct type under nominal typing; and a type alias for `TypeA` (like `typedef` in C) under structural typing.

To submit your answer, please fill in the file *exercise5/task1.csv*. One row corresponds to one code snippet. The second column corresponds to nominal typing. The third column corresponds to structural typing. Fill in `valid` if the code snippet under the corresponding type system is well typed, and `invalid` if it is not well typed. The first example above is well-typed under both nominal and structural typing, which is why the second line of the template CSV file is `example, valid, valid`.

Code Snippet 1

```
1 let a: String = "somestring";
2 let b: String = a;
```

Code Snippet 2

```
1 type Meters = { value: Integer }; // Using a wrapper type for checking units.
2 let a: Meters = { value: 42 };
3 let b: Integer = a;
```

Code Snippet 3

```
1 type Meters = { value: Integer };
2 type Liters = { value: Integer };
3 let a: Meters = { value: 42 };
4 let b: Liters = a;
```

Code Snippet 4

```
1 type Meters = { value: Integer };
2 type MyMeters = Meters;
3 let a: Meters = { value: 42 };
4 let b: MyMeters = a;
```

Code Snippet 5

```
1 type StudentProps = { first: String, last: String };
2 type Student = { id: Integer, props: StudentProps };
3 type CourseProps = { roomNumber: Integer, name: String };
4 type Course = { id: Integer, props: CourseProps };
5 let a: Student = { id: 1337, props: { first: "John", last: "Haxor" } };
6 let b: Course = a;
```

Code Snippet 6

```
1 type StudentName = String;
2 type Student = { id: Integer, name: StudentName };
3 type CourseName = String;
4 type Course = { id: Integer, name: CourseName };
5 let a: Student = { id: 1337, name: "John␣Haxor" };
6 let b: Course = a;
```

Evaluation Criteria: Your solution will be compared against the correct result of type checking each code snippet under the two given type system choices.

2 Task II (20% 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).

$$
\begin{array}{rr}
e ::= \texttt{true} \mid \texttt{false} & \text{boolean literals} \\
\mid n & \text{integer literals, so } n \in \mathbb{N} \\
\mid \texttt{!}e & \text{boolean negation} \\
\mid e \;\&\&\; e & \text{boolean conjunction} \\
\mid e + e & \text{integer addition} \\
\mid e = e & \text{equality} \\
\mid \texttt{if } e \texttt{ then } e \texttt{ else } e & \text{if-then-else}
\end{array}
$$

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

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

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

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

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

(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
- `"hyptoheses"`: 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:

$$\dfrac{\dfrac{}{\texttt{false} : Bool}\ \text{T-False}}{\texttt{!false} : Bool}\ \text{T-Not}$$

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",
  "type": "Bool",
  "rule": "T-Not",
  "hypotheses": [ {
      "expression": "false",
      "type": "Bool",
      "rule": "T-False",
      "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 in the `expression` property will not matter. 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 do so for the following expressions:

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

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.

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

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

Options for code fragments to insert:

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

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 **memory representation and alignment** of structs in C (and other "systems" languages). Given several definitions of structs in C and different alignment requirements, you should compute the **most compact** memory layout of the struct that adheres to the requirements.

You have to compute the memory layouts under four different sets of alignment requirements, as shown in Table 1. For this task assume 8 bit `chars`, 32 bit `ints`, and 32 bit `floats`. The first offset in each struct is 0. An alignment requirement of 4 bytes means that the begin byte offset of that field must divide without remainder by 4. If reordering is allowed, fields in the memory layout *can* appear in a different order than in the struct declaration in the C source code, in order to get a more compact memory representation of the total struct. (Note: For historical reasons, real C compilers do not reorder fields. But, e.g., in Rust reordering of struct fields is allowed and done in practice by the compiler.)

	Alignment Requirements			
	R1	R2	R3	R4
`char` fields	no alignment	8 bytes	4 bytes	4 bytes
`int` fields	no alignment	8 bytes	4 bytes	4 bytes
`float` fields	no alignment	8 bytes	8 bytes	8 bytes
reordering of fields	not allowed	not allowed	not allowed	allowed

Table 1: Four different sets of alignment requirements for different datatypes and if reordering is allowed.

You have to compute the memory layouts for the following three struct definitions:

Struct A

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

Struct B

```c
struct B {
  int  field1;
  char field2;
  int  field3;
  char field4;
};
```

Struct C

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

For solving the task, it may be useful to draw layout figures with pen and paper. 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. Fill in the offset (i.e., the byte index at which the field begins) in the columns corresponding to each of the four alignment requirements.
To give an example: Under the first alignment requirements R1, `structA` is layed out as follows. `field1` starts at byte offset 0, then directly followed without padding by `field2` at byte offset 1 (since there is no alignment requirement for `chars` and since they take 1 byte each), and finally the `int field3` at byte offset 2 (again, because there is no alignment requirement for `ints`). 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.

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[2]) and local variables (since Python 3.6[3]). 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. Note that the types also support generics, e.g., `List[str]` is a valid type.

Evaluation Criteria: Your code will be evaluated by type-checking it with `mypy` version 0.560 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.

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