Programming Language Design
Assignment 3 2022

Torben Mogensen and Hans Hüttel

6th March 2022

This assignment is *individual*, so you are not allowed to discuss it with other students. All questions should be addressed to teachers and TAs. If you use material from the Internet or books, cite the sources. Plagiarism *will* be reported.

Assignment 1 counts 20% of the grade for the course, but you are required to get at least 33% of the possible score for every assignment, so you can not count on passing by the later assignments only. You are expected to use around 9 hours in total on this assignment, including the resubmission (if any). Note that this estimate assumes that you have solved the exercises suggested in slides/videos and participated in the plenary and TA sessions. If not, you should expect to use considerably more time for this assignment.

The deadline for the assignment is **Friday March 18 at 16:00 (4:00 PM)**. The individual exercises below are given percentages that provide a rough idea how much they count (and how much time you are expected to use on them). These percentages do *not* translate directly to a grade, but will give a rough idea of how much each exercise counts.

In normal circumstances, feedback will be given by your TA no later than 27 March. Note that, even if the TAs find no errors in your submission, this is no guarantee that it is perfect, so we strongly recommend that you take time to improve your submission for resubmission regardless of the feedback you get. You can do so until **29 March at 16:00 (4:00 PM)**. Note that resubmission is made as a separate mandatory assignment on Absalon. If you resubmit, the resubmission is used for grading, otherwise your first submission will be used for grading.

The assignments consists of several exercises, some from the notes and some specified in the text below. You should hand in a single PDF file with your answers and a zip-file with your code. Hand-in is through Absalon. Your submission must be written in English.

A3.1) (15%) Java allows for polymorphism in the form of generic classes that are parameterized with type variables. See e.g. `https://www.tutorialspoint.com/java/java_generics.htm` for an explanation of this.

Here is a Java program that the Java compiler rejects.

```
class Bingo<T> {
public void dingo(T t) {
        System.out.println(t.bingoString());
    }
}
```

a) Why does the Java compiler reject the program?

b) If the code above were allowed to compile without errors, what might go wrong? Give a snippet of code that uses the Bingo class to cause a run-time error.

c) Now fix the code such that it will compile and preserves its functionality. You should not modify the method dingo, but you can change or add any code. Your solution should include all details needed to check that Bingo is a valid Java class.

d) Assume that Bingo was fixed to resolve the problem we saw before. Let A and B be two classes where A is a supertype of B and Bingo<A> and Bingo<B> are valid instantiations. Consider the following method:

```
void frank(Bingo<A> bingoA) {
  Bingo<B> bingoB = bingoA;
  bingoB.dingo(new B());
}
```

The Java compiler also rejects this code. Is the code safe? That is, if it were allowed to compile, would it lead to a run-time error?

e) Now let us relax the Java type system rules such that it would allow generic types to be *contravariant*.

- Will the method frank compile now?
- What are the situations that require dynamic checks in order to enable contravariant generics in a language without limiting what one can write in a generic class?

A3.2) (15%) In many languages, all clients of a module have the same view of the module, but in the Eiffel language the export statement allows a module to restrict which entities it exports to whom.

Here is an example.

```
export entity1{M1}, entity2{M1,M2}, entity3{M3}, entity4
```

says that module M1 has access to entity1, entity2 and entity4, whereas module M2 has access to entity2, entity3 and entity4. Every module can import and use entity4.

In Modula-2 one can instead use selective imports. To selectively important entities from a module one writes

```
from module import entity-list;
```

where module is the name of the module that one wants to import entities from and entity−list is the list of entities that one wants to import from it. No other entities from the module become available afterwards.

a) What are the advantages and disadvantages of the approach taken in Eiffel?

b) What are the advantages and disadvantages of the approach taken in Modula-2?

A3.3) (20%) Here is a Haskell function.

```
plop [] w = [w]
plop ((u,v):uvs) w = u:v: (plop uvs w)
```

a) Explain as precisely as possible what plop computes.

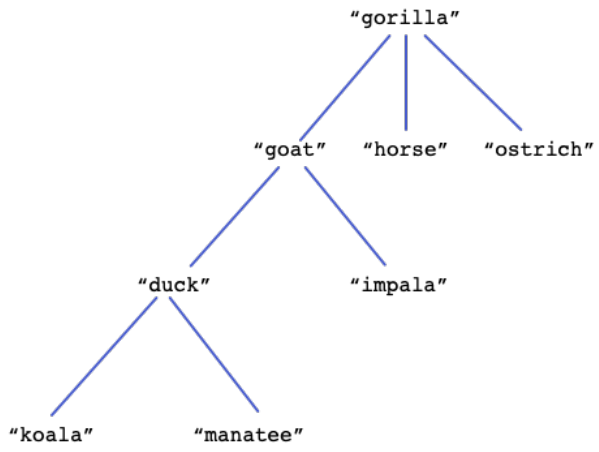
b) Use the Hindley-Milner type inference algorithm to find the type of plop and explain the steps shown in Section 8.4.5 that you used to find the type. Remember to show the intermediate results, and to explain how you arrived at them.

A3.4) (20%) A *mixed tree* is a tree structure in which every node is labelled with a string and has either three, two or no children. Here are three Prolog clauses defining mixed trees.

```
mixed(root(X,T1,T2,T3)) :- string(X), mixed(T1), mixed(T2), mixed(T3).
mixed(root(X,T1,T2)) :- string(X), mixed(T1),mixed(T2).
mixed(root(X)) :- string(X).
```

a) Explain why and in what way mixed and root are different notions.

b) Here is a picture of a mixed tree, mytree.



Represent this mixed tree as a term in Prolog.

c) The leftmost element of a mixed tree is the string that labels the leaf found by taking the leftmost branch of the tree. In the tree shown above, the leftmost element is "koala".

Give the clauses defining a Prolog predicate leftmost(M,E) which holds if E is the leftmost element of the mixed tree M. Implement this in SWI-Prolog.

d) Describe precisely what will happen when we give SWI-Prolog the query ?− leftmost (..., X).

where in the query ... is replaced by the term describing mytree that you found previously.

A3.5) (15%) Unification is important in different settings in the world of programming language techniques. So far we have seen it in the settings of Hindley-Milner type inference and in Prolog.

Below are four instances of Hindley-Milner type expressions T_1 and T_2 that we would like to unify. The lowercase t's denote type variables. For each of these four instances, find out if T_1 and T_2 can be unified and, in the cases where they can, provide a unifying substitution. In the cases where T_1 and T_2 cannot be unified, give an explanation of how and where unification fails.

$$T_1 = \texttt{real} \to t_4 \text{ and } T_2 = t_5 \to (\texttt{string} \to \texttt{int})$$
$$T_1 = t_1 \to (t_2 \to \texttt{real} \text{ and } T_2 = \texttt{int} \to ((t_3\ \texttt{list}) \to \texttt{real})$$
$$T_1 = (\texttt{int}\ \texttt{list})\ \texttt{list} \text{ and } T_2 = t_6\ \texttt{list}$$
$$T_1 = ((t_7\ \texttt{list} \times t_8) \times t_8 \text{ and } T_2 = (t_9 \times (t_10\ \texttt{list}) \times t_9$$

A3.6) (15%) Sometimes it seems as if languages that have a static type system also come equipped with static scope rules. There are well-known examples of dynamically typed languages with static scope rules. But suppose we want the opposite: A static type system for a language with dynamic scope rules. What are the challenges here? How should we deal with them?

Start by considering the follow code snippet written using C-like syntax.

```
int plip(int y) {
    return x + y;
}

int mango(int y) {
    double x = 3.0;
    return plip(y);
}
```

Assume that variables are declared with a type when they are introduced. Suggest restrictions on how variables are to be used or declared that will ensure that static type checking is possible and explain which issues these restrictions avoid.