Type Soundness & Subtyping

Exercise 1

Consider a simple programming language with integer arithmetic, boolean expressions and user-defined functions.

```
T := Int | Bool | (T₁, …, Tₙ) => T
t := true | false | c₁
   | t₁ == t₂ | t₁ + t₂ | t₁ && t₂
   | if (t₁) t₂ else t₃
   | f(t₁, … , tₙ) | x
```

Where c_1 represents integer literals, == represents equality (between integers, as well as between booleans), + represents the usual integer addition and && represents conjunction. The meta-variable `f` refers to names of user-defined function and `x` refers to names of variables.

You may assume that you have a fixed environment `e` which contains information about user-defined functions (i.e. the function arguments, their types, the function body and the result type).

Part 1

Write down the *"usual"* typing rules for this language.

Part 2

Inductively define the substitution operation for your terms, which replaces every free occurence of a variable in an expression by an expression *without free variables*.

Prove that substitution preserves the type of an expression, given that the variable and the expression have the same type.

Part 3

Write the operational semantics rules for the language, assuming ***call-by-name*** semantics for function calls. In call-by-name semantics, the arguments of a function are not evaluated before the call. In your operational semantics, parameters in the function body are to merely be substituted by the corresponding unevaluated argument expression.

Part 4

Adapt the soundness proof seen in the last lecture to account for the new semantics. Prove only the cases related to function application.

Exercise 2

In this second exercise, we will have a look at a simple programming language with the following types and terms:

```
T := Integer | Pos | Neg
t := c₁ | t₁ + t₂ | t₁ * t₂ | t₁ / t₂
```

`Integer` is the type of all integer numbers, while `Pos` is the type of all *strictly* positive integer numbers and `Neg` the type of all *strictly* negative numbers. Note that, interestingly, some terms will accept multiple types.

For instance, `14` will have the types `Integer` and `Pos`, while `-2` will have the types `Integer` and `Neg`. The constant `0` on the other hand will only have the type `Integer`.

Part 1

Write down typing rules for the terms of the language. Try to preserve information about positivity and negativity. Also, make sure that your type system prohibits division by zero.

Part 2

Under your type system, what are the types, if any, of the following terms? Write down a derivation for each possible type.

```
1 + 1

-2 * 4

-1 * (2 + -1)

7 / (18 + -1)
```

Part 3

We now introduce a new relation, T_1 `<:` T_2, which we call the *subtyping* relation.

T_1 `<:` T_2 can be read as "T_1 is a subtype of T_2". When T_1 `<:` T_2, terms of type T_1 can safely be used in the context where terms of type T_2 are expected. In this exercise, what pairs of types can be made part of this subtyping relation? List all such possible pairs.

Part 4

Write down the *subsumption* rule, which bridges the gap between the subtyping relation and the typing relation. The rule should state that if a term has a type T_1 and T_1 is a subtype of T_2, then the term has also type T_2.

Now that you have defined this rule, can you remove some of the typing rules you had previously defined for the various constructs of the language ?

Part 5

Let's now expand our language and add a primitive "`power`" function to it:

t := ... | power(t_1, t_2)

With the following typing rule:

$$\frac{\Gamma \vdash t_1 : \texttt{Integer} \qquad \Gamma \vdash t_2 : \texttt{Integer}}{\Gamma \vdash \texttt{power}(t_1,\ t_2) : \texttt{Integer}}$$

Typecheck the following expression under the empty environment. Show a type derivation.

```
power(7 / 2, 3)
```

Does there exist multiple valid type derivations that assign the same type to the above expression?