- Prepare your solutions on paper.
- Marking an exercise in OLAT means that a significant part of that exercise has been treated.
- Upload your Haskell files and your paper solution in OLAT, the latter as one PDF.

Exercise 1 *Matching Algorithm with Error Monad* **4 p.**

Reconsider the matching algorithm of slide 3/23 that was already implemented on exercise sheet 5.

1. Change the existing implementation so that `match` has type `Term -> Term -> Check Subst` where the error message should indicate why the matching failed. (here, `Check a` should be defined as `Either String a`) (2 points)
2. The problem with the return type `Check Subst` is that it is tedious to continue to work with, e.g., if one wants to continue in different ways, depending on why the matching algorithm failed.

   A more convenient error type might be the following one, which clearly indicates the error via a dedicated type that additionally stores the information on why it clashes, e.g., there was a clash between symbol f and g.

   ```
   data Match_Error = Fun_Var FSym Var | Fun_Clash FSym FSym | Var_Clash Var Term Term

   type Match_Result a = Either Match_Error a
   ```

   Adjust your algorithm so that `match` has return type `Match_Result Subst`. What do you need to change? (2 points)

Exercise 2 *Type-inference* **6 p.**

Consider the type-inference algorithm on slide 4/8.

1. Show that the modified completeness property

   $$\text{if } t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau \text{ then } \mathit{infer\_type}\ \Sigma\ \tau\ t \neq \mathit{failure}$$

   cannot be directly proven by induction on t. To this end, try to prove exactly this property by induction on t and illustrate that one of the cases is not provable, since the IH is too weak. (2 points)
2. Prove the correct completeness statement:

   $$\text{if } t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau \text{ then } \mathit{infer\_type}\ \Sigma\ \tau\ t = \mathit{return}\ (\mathcal{V} \cap \mathcal{V}ars(t))$$

   You don't have to be formal, but should just indicate the crucial reasoning steps. You can use properties of auxiliary functions (*distinct*, *nub*, *concat*, etc.) without having to prove these. (4 points)

Exercise 3 *Processing Function Definitions* **10 p.**

Slide 4/21 contains a Haskell function to process data definitions. The task of this exercise is to implement a similar function for checking and processing function definitions w.r.t. slide 3/15.

1. Implement a Haskell function `linear :: Term -> Bool` which decides whether a term is linear or not, cf. slide 3/14. (2 points)

2. Implement a Haskell function

```
check_equation ::
  Sig_List ->       -- defined symbols, including f
  Sig_List ->       -- constructors
  FSym ->           -- f
  FSym_Info ->      -- type of f
  (Term, Term) ->   -- equation (l,r)
  Check ()
```

   that checks whether a single equation satisfies the conditions that are mentioned on slide 3/15. Of course, you should use the provided functions for type-checking, type-inference, etc., as much as possible. (5 points)

3. Implement the Haskell function `process_function_definition` mentioned on slide 4/23. (3 points)

Once you have completed your implementation, you can test it via `test`, which processes some example program, which should be accepted.
By manually inserting errors into the example program, you can run `test` again, to see whether these errors are detected by your implementation.