universität innsbruck

Program Verification	SS 2024	LVA 703083+703084
Sheet 4		Deadline: April 16, 2024, 3pm

- Prepare your solutions on paper.
- Mark the exercises in OLAT before the deadline.
- Upload your Haskell solution in OLAT.
- Marking an exercise means that a significant part of that exercise has been treated.

Exercise 1 *Pattern Disjointness* **11 p.**

Consider the definition of pattern disjointness on slide 3/39. Testing whether a program is pattern disjoint is not directly possible based on this definition, since it involves a quantification over infinitely many terms. In this exercise, the aim is to develop an algorithm to decide whether a program is pattern disjoint, based on unification.
Unification of two terms s and t is the question whether there exists a substitution σ such that $s\sigma = t\sigma$ and deliver such a substitution in case it exists. So in contrast to matching, here the substitution is applied on both terms.
A concrete unification algorithm is due to Martelli and Montanari, cf. `https://en.wikipedia.org/wiki/Unification_(computer_science)#Unification_algorithms`, and its structure is quite similar to the matching algorithm.

1. Task: have a look at this algorithm and apply it step-by-step on $s := \mathsf{append}(\mathsf{Cons}(x, xs), ys)$ and $t := \mathsf{append}(xs, \mathsf{Nil})$. (2 points)
2. Consider the following algorithm to decide pattern disjointness of a program:

   check for each pair of distinct equations $\ell_1 = r_1$ and $\ell_2 = r_2$ that ℓ_1 and ℓ_2 do not unify.

   Argue that this algorithm is not correct with the help of the following functional program (datatype definitions omitted). Here, you don't have to perform the unification algorithm step-by-step.

   $$\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys)) \quad (1)$$
   $$\mathsf{append}(\mathsf{Nil}, \mathsf{Cons}(y, ys)) = \mathsf{Cons}(y, ys) \quad (2)$$
   $$\mathsf{append}(xs, \mathsf{Nil}) = xs \quad (3)$$

   (3 points)
3. Identify the problem and slightly adjust the previous algorithm so that it indeed decides pattern disjointness. (3 points)
4. Prove soundness of your algorithm. (3 points)
5. Prove completeness of your algorithm. This is a bonus exercise which is worth additional 4 points.

Exercise 2 *Processing Function Definitions* **9 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

   ```
   checkEquation ::
     SigList ->          -- defined symbols, including f
     SigList ->          -- constructors
     FSym ->             -- f
     FSymInfo ->         -- 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. (4 points)

3. Implement the Haskell function `processFunctionDefinition` 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.