Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exam Sheet

August 28-30, 2023

How to do this exam

- You can earn 50 points from the questions on this sheet.
- You can earn 150 points from the programming task.
- You need ≥ 100 points to pass the exam and you must earn ≥ 20 points from this sheet. Otherwise you are free to choose.
- Do not change the names and types of the functions as given on this sheet and in the template file supplied. You may add further imports as long as they do not require adding new files.

In the lecture, we defined signatures and terms over a set of variables. Consider the following datatypes for representing terms, positions in a term, substitutions, and signatures. Function symbols as well as variables are represented by `Ident`.

```
import qualified Data.Map as M

type Ident        = Char
data Term         = Term { symbol :: Ident, subterms :: [Term] }
                  | Var { name :: Ident }
                  deriving (Eq)
type Pos          = [Int]
type Substitution = M.Map Ident Term
type Signature    = M.Map Ident Int
```

Question 1 (Terms, 30 points, file: Questions.hs)

(a)
```
applySubst :: Substitution -> Term -> Term
applySubst subst t = ...
```

A substitution maps a variable to a term. Applying a substitution `subst` to term `t` means to traverse the term and replace each occurrence of a variable like `Var 'A'` by the term returned by the substitution for `'A'`. Variables that are not in the domain of the substitution stay as they are.

Examples

```
λ> tA = Term 'a' []
λ> tB = Term 'b' []
λ> tC = Term 'c' [tA, tB]
λ> tD = Term 'd' [Var 'A', Var 'C', Var 'X']
λ> sub = M.fromList [('A', tA), ('C', tC)]
λ> applySubst sub tD == Term 'd' [tA, tC, Var 'X']
True
```

(b)
```
matchHere :: Term -> Term -> Maybe Substitution
matchHere pattern subject = ...
```

The function call `matchHere pattern subject` checks if the `pattern` term occurs at the root of the `subject` term. If the pattern contains variables, then the corresponding term in the `subject` can be arbitrary. Assume that the `pattern` is *linear,* that is, each variable occurs at most once in it.

The function returns

- `Just subst`, if the pattern matches at the root of the `subject`. The returned substitution fulfills `applySubst subst pattern == subject`
- `Nothing`, if the pattern does not match at the root.

Examples

```
λ> matchHere tA tA == Just M.empty
True
λ> matchHere tA tB == Nothing
True
```

(c)
```
findLeftmostMatch :: Term -> Term -> Maybe (Pos, Substitution)
findLeftmostMatch pattern subject = ...
```

The function call `findLeftmostMatch pattern subject` returns the leftmost position in `subject` where `pattern` occurs. It either returns

- `Just (pos, subst)`, a pair of a position and a substitution such that `matchHere pattern (subject `at` pos) == Just subst`; or
- `Nothing`, if no such position exists.

Here is a definition of the `at` function that extracts the term at a given position from another term. It assumes that the position argument is valid for the term.

```
at :: Term -> Pos -> Term
at t (i : p) = at (subterms t !! i) p
at t [] = t
```

Examples

```
λ> pC = Term 'c' [Var 'X', Var 'Y']
λ> findLeftmostMatch tA tC == Just ([0], M.empty)
True
λ> findLeftmostMatch pC tA == Nothing
True
λ> findLeftmostMatch pC tC == Just ([], M.fromList [('X', tA), ('Y', tB)])
True
```

Question 2 on the next page

Question 2 (QuickCheck Generators, 20 points, file: Questions.hs)

(a)
```
genTerm :: Signature -> Gen Term
genTerm sig = ...
```

The heart of the QuickCheck library is the generation of test data from types. The function `genTermSimple sig` takes a term signature `sig` and returns a generator for the type `Term`.

Implement this generator such that

- variable identifiers are uppercase ASCII letters,
- all generated terms adhere to the signature `sig`,
- all function symbols contained in the signature `sig` can appear in the generated terms,
- the generator always terminates.

Every QuickCheck generator is equipped with a size, which can be queried and modified using the `sized`, `getSize`, `resize`, and `scale` functions from the QuickCheck library. Use the generator's size to ensure termination. The concrete interpretation is up to you but larger sizes should, on average, result in larger terms.

(b)
```
genTermVarProb :: Int -> Signature -> Gen Term
genTermVarProb varP sig = ...
```

The generator `genTermVarProb varP sig` generates the same kind of terms as `genTerm sig`, but the extra argument `varP` defines the probability (in percent) that the generated term is a variable. (Most of the code is the same as for `genTerm`, so you only get points for changes that affect the probability in the requested way.)