Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
22.01.2021
Deadline: 01.02.2021, 23:59

Functional Programming and Verification

Sheet 11

Tutorial Exercises

Exercise T11.1 Abstract Data Types: Maps

Note: Please use the templates `AssocList.hs` and `AssocListTests.hs` provided on moodle for this exercise.

You have already seen association lists `Eq k => [(k,v)]` as a way to represent maps with keys k und values v. In order to prevent user from creating invalid association lists (e.g. containing multiple values for some key), we want to hide the implementation in a module.

1. Define a module `AssocList` that only exports a type `Map k v` and the following functions:

```
newtype Map k v = ...
empty :: Map k v
insert :: Eq k => k -> v -> Map k v -> Map k v
lookup :: Eq k => k -> Map k v -> Maybe v
delete :: Eq k => k -> Map k v -> Map k v
keys :: Map k v -> [k]
```

   Calling `insert` with an existing key should replace the associated value. Internally, the maps should be represented using association lists.

   Note: Prelude also exports a function `lookup`. To prevent naming conflicts, you can hide this import using `import Prelude hiding (lookup)`.

2. Define a function `invar :: Eq k => Map k v -> Bool` in `AssocList` that checks whether the map does not contain duplicate keys. Then define QuickCheck properties in a separate module that check whether `invar` is invariant under all functions returning a map as discussed in the lecture (slide 340).

   Note: To check your properties, say `prop_invarInsert`, you need to explicitly tell QuickCheck the types of the values to generate. For example:

```
quickCheck (prop_invarInsert :: Int -> String ->
              AL.Map Int String -> Property)
```

3. We next check if our implementation (imported as `AL.Map`) behaves correctly when compared to the `Map` datatype provided by `Data.Map` from the package `containers` (imported as `DM.Map`). Define a function `hom :: Ord k => AL.Map k v -> DM.Map k v` that transforms our maps to the one provided by the containers library. Then check whether AL.Map simulates DM.Map by defining QuickCheck properties for every function in `AssocList` as discussed in the lecture (slide 340).

Exercise T11.2 Substitute Teacher

The following datatype represents a simplified version of Haskell:

```
data Term = Var String | Abs String Term | App Term Term
```

A program in this language is an instance of the datatype `Term` which is either

- just a variable, e.g. `Var x`, which corresponds to a function or a constant `x`,
- an anonymous function definition `Abs x t`, which corresponds to the lambda expression `(\x -> t)` for a term `t`, or
- a function application `f x` where both `f` and `x` are terms themselves.

We distinguish between free and bound variables. A variable is bound if there exists an enclosing lambda abstraction that binds it. All other variables are free, e.g. in the term `(\x -> x y) z`, `x` is bound while `y` and `z` are free. Note that bound variable names are interchangeable whereas this is not the case for free variables. For example, the terms `(\x -> x y)` and `(\z -> z y)` are equal while the terms `(\x -> x y)` and `(\x -> x z)` are not equal. On the above datatype, implement the following functions:

1. Define `freeVars :: Term -> [String]` which collects all free variables in a term.
2. In order to have a programming language, we need an evaluation function for our terms. To this end, implement a function `substVar :: String -> Term -> Term -> Term` where `substVar x r t` substitutes all free occurences of `Var x` in the term `t` by the term `r`. Now, we can perform a single evalution step by evaluating a function application `(\x -> t) r` to `substVar x r t`.

   Important: the function `substVar` makes a key assumption about `x`, `t` and `r`. To find out what the assumption is, think about what happens when substituting `x` with the variable `y` in the equivalent terms `(\y -> x y)` and `(\z -> x z)`.
3. (Bonus) Implement a capture-avoiding version of `substVar`.

Note: the above programming language is commonly known as λ-calculus and forms the basis of most functional programming languages.

Homework

You need to collect 3 out of 4 points (P) to collect a coin.

Exercise H11.1 Abstract Data Types: Graphs [1P]

In this exercise you will implement a module for directed graphs. It is up to you to decide how you represent the graphs internally. Note that you may not use the `containers` package in this exercise.

Define a module `Graph` that only exports a type `Graph n` and the following functions:

```
newtype Graph n = ...
empty :: Graph n
nodes :: Eq n => Graph n -> [n]
addEdge :: Eq n => (n,n) -> Graph n -> Graph n
fromEdgeList :: Eq n => [(n,n)] -> Graph n
neighbors :: Eq n => n -> Graph n -> [n]
isReachable :: Eq n => n -> n -> Graph n -> Bool
transpose :: Eq n => Graph n -> Graph n
```

`empty` should return an empty graph.

`nodes g` should return alist of all nodes in `g`.

`addEdge (n,m) g` adds a directed edge from `n` to `m` to the graph `g`. If a node is not yet part of the graph, it is added implicitly by this function.

`fromEdgeList es g` adds the edges in the list `es` to an initially empty graph.

`neighbors n g` returns all nodes that are connected to `n` via an edge. If `n` is not a node in the graph, it returns `[]`. Make sure that the result does not contain duplicate nodes.

`isReachable n m g` returns `True` iff there is a path from `n` to `m` in `g`.

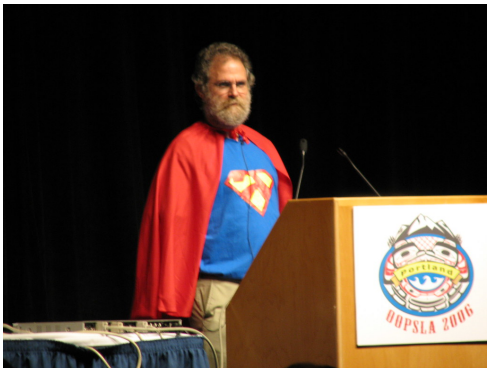
`transpose g` returns a graph where the direction of all edges in `g` has been reversed.

Note: In preparation for the exam, it might be a good exercise to implement some tests of your graph module against an existing graph library, e.g. `Data.Graph`.

Exercise H11.2 Capture Me If You Can! [1P + 1P + 1P]

When implementing `substVar` in the the tutorial, we were faced with the problem of variable capture due to name clashes. Consider the example from the tutorial exercise: if we substitute `x` by `y` in the equivalent terms `(\y -> x y)` and `(\z -> x z)`, we obtain the terms `(\y -> y y)` and `(\z -> y z)`, respectively. Those two terms are not equivalent anymore, e.g. evaluating `(\y -> y y) x` and `(\z -> y z) x` gives different results.

One way to work around this problem is to make the substitution function capture avoiding, i.e. if a free variable that we substitute into a term would be captured by an enclosing λ-abstraction, we instead rename the variable that the λ-abstraction binds. In the example `(\y -> x y)`, we



Philipp Wadler (designer of Haskell) in his legendary lambda calculus superman costume

could rename `y` to `z` which gives us `(\z -> x z)`. Now we can substitute `x` by `y` without variable capture.

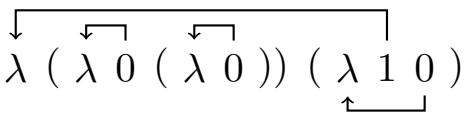
Since implementing capture-avoiding substitution is quite tricky, we will instead change the representation of terms. In this representation, we explicitely distinguish between free and bound variables which eliminates the possibility of accidentally binding free variables:

```
infixr 5 :$:
data BTerm = Free String | Bound Int |
             BAbs BTerm | BTerm :$: BTerm
```

The key difference is that we now have a *nameless* representation of bound variables where the number of a bound variable `Bound i` tells us by which of the enclosing λ-abstraction that variable is bound. More specifically, a bound variable `Bound i` is bound by the i-th enclosing lambda abstraction. In other words, you have to go up i abstractions to find the binding λ-abstractions if you view an instance of `BTerm` as a tree. To improve readability, we provide you with an instance `Show BTerm` that invents names (prefixed by `_`) for bound variables. In order to understand how binding of variables works, consider the picture below that illustrates the binders of the term

```
BAbs $ (BAbs $ Bound 0 :$: (BAbs $ Bound 0)) :$:
       (BAbs $ Bound 1 :$: Bound 0).
```

Note that `$` refers to the standard function application operator in Haskell and `:$:` to the function application *constructor* that we defined with the datatype `BTerm`.



Implement the following functions:

1. Define a function `fromTerm :: Term -> BTerm` that turns terms as introduced in the tutorial into terms as explained above.

2. Define a function `reduce :: BTerm -> BTerm` that evaluates the function application `BAbs t :$: r`, i.e. that replaces all variables that are bound by the abstraction `BAbs` by the term `r`.
3. Using the previous function, implement a function `reduceFull :: BTerm -> BTerm` that evaluates function applications of the form `BAbs t :$: r` as long as the term contains any.

If you implemented the above functions, then congratulations, you now have a Turing-complete programming language. To try out your shiny new language, the template provides you with functions to convert between integers and Church numerals, which were already introduced in tutorial exercise 6.3. For example, try

```
fromChurchNum $ churchAdd (churchNum 10) (churchNum 12)
```

> Nenn es dann, wie du willst,
> Nenn's Glück! Herz! Liebe! Gott
> Ich habe keinen Namen
> Dafür! Gefühl ist alles;
> Name ist Schall und Rauch,
> Umnebelnd Himmelsglut.
>
> — Goethe's Faust