Types

Jonas Hübotter

February 13, 2022

1 Exercises

1.1 Type Classes

1. [endterm 2020] We define a typeclass of integer containers as follows:

```
class IntContainer c where
  -- the empty container
  empty :: c
  -- insert an integer into a container
  insert :: Integer -> c -> c
```

Moreover, we define an extension of integer containers called `IntCollection` as follows:

```
class IntContainer c => IntCollection c where
  -- the number of integers in the collection
  size :: c -> Integer
  -- True if and only if the integer is a member of the collection
  member :: Integer -> c -> Bool
  -- extracts the smallest number in the collection
  -- if such a number exists.
  extractMin :: c -> Maybe Integer
  -- "update f c" applies f to every element e of c.
  -- If "f c" returns Nothing, the element is deleted;
  -- otherwise, the new value is stored in place of e.
  update :: (Integer -> Maybe Integer) -> c -> c
  -- "partition p c" creates two collections (c1 , c2) such that
  -- c1 contains exactly those elements of c satisfying p and
  -- c2 contains exactly those elements of c not satisfying p.
  partition :: (Integer -> Bool) -> c -> (c,c)
```

Assume there is a type `data C` with a corresponding `IntContainer` instance. Moreover, assume you are given the following function:

```
-- "fold f acc c" folds the function f along c (in no particular order)
-- using the start accumulator acc.
fold :: (Integer -> b -> b) -> b -> C -> b
```

Define an instance `IntCollection C`.

1.2 Algebraic Data Types

1. [endterm 2014] A *Collatz* sequence is a special sequence of natural numbers. The element c_{k+1} is obtained from the previous element c_k as follows:

$$c_{k+1} = \begin{cases} \frac{c_k}{2} & \text{if } c_k \text{ even} \\ 3 \cdot c_k + 1 & \text{if } c_k \text{ odd} \end{cases}$$

The first element c_0 can be any natural number $n > 0$. The sequence ends when the number 1 is reached.

Example: let $c_0 = 6$. The entire sequence then is $6, 3, 10, 5, 16, 8, 4, 2, 1$.

   (a) Define a function `collatz :: Integer -> [Integer]` that given an initial value returns the Collatz sequence as a list.

   (b) Implement the function `unfold :: (a -> Maybe a) -> a -> [a]`. For a call `unfold f a` the function f should be repeatedly applied to a and stop with a result of `Nothing`. The return value is a list of all intermediate results, including the initial value of a.

   Example: For $f a_0 =$ `Just` a_1, $f a_1 =$ `Just` a_2, $f a_n =$ `Nothing` we have `unfold` $f a_0 = [a_0, a_1, \dots, a_n]$.

   (c) Find a function `next` such that `collatz n = unfold next n` for $n > 0$.

2. [sheet 8] Trees.

   (a) In a binary tree, we can only descend to the left or to the right. Define a data type `Tree` with constructors `Leaf` and `Node` for binary trees.

   (b) Implement a function `sumTree :: Num a => Tree a -> a` that returns the sum of all values in a tree.

   (c) Implement a function `cut :: Tree a -> Integer -> Tree a` that cuts off a tree after a given height.

   (d) Implement a function `foldTree :: (a -> b -> b) -> b -> Tree a -> b` that folds a function over a tree. The fold should process the right children of a node first, then the node itself, and lastly the left children.

   (e) Use `foldTree` to implement a function `inorder :: Tree a -> [a]` that returns all elements of a tree in left to right order.

   (f) Use `foldTree` to implement a function `findAll :: (a -> Bool) -> Tree a -> [a]` that returns all elements of a tree that satisfy a given predicate.

3. [endterm 2020] We define the following types:

   - An *atom* is either `F` (falsity), `T` (truth), or a variable:

```
type Name = String
data Atom = F | T | V Name deriving (Eq, Show)
```

   - A *conjunction* is an atom or the conjunction of two conjunctions:

```
data Conj = A Atom | Conj :&: Conj deriving (Eq, Show)
```

   (a) Write a function `contains :: Conj -> Atom -> Bool` such that `contains c a` returns `True` if and only if `a` occurs in `c`.

   (b) Write a function `implConj :: Conj -> Conj -> Bool` such that `implConj c1 c2` returns `True` if and only if conjunction `c1` logically implies conjunction `c2`. For example:

```
A F `implConj` c = True -- for any conjunction c
c `implConj` A T = True -- for any conjunction c
A (V "v") `implConj` A (V "v") = True
A (V "v") `implConj` A (V "v") :&: A (V "w") = False
A (V "w") :&: A (V "v") `implConj` A (V "v") :&: A (V "w") = True
```

4. [sheet 12] We consider a simple programming language, namely λ-calculus. It is the basis for most functional programming languages including Haskell. A program in λ-calculus is built up from terms, which are either

   - just a variable, e.g. x,
   - an anonymous function defintion $(\lambda x.\ T)$, which binds a variable x in term T, or
   - a function application $T\ U$ where both T and U are terms themselves.

   Note that bound variable names are interchangeable whereas this is not the case for free variables. For example, the terms $(\lambda x.\ x\ y)$ and $(\lambda z.\ z\ y)$ are equal while the terms $(\lambda x.\ x\ y)$ and $(\lambda x.\ x\ z)$ are not equal. Start by defining a datatype `Term` in Haskell that models the λ-calculus using strings for variables names. On this datatype, implement the following functions:

   (a) Instantiate `Show` for `Term` by representing variables as just their name, λ-abstractions as `(\x -> T)` like in Haskell and use spaces for function application.

   (b) Define `freeVars :: Term -> [String]` which collects all free variables in a term, i.e. variables that are not bound by an enclosing λ-abstraction.

   (c) Implement a function `substVar :: String -> Term -> Term -> Term` which, when called with `substVar x r t`, replaces all free occurences of `x` in `t` by the term `r`. *Important:* the function `substVar` makes a key assumption about the terms `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 $(\lambda y.\ x\ y)$ and $(\lambda z.\ x\ z)$.

5. [sheet 11] In this exercise we will work with the datatype `Either` from the Prelude, which is defined as follows:

   ```
   data Either a b = Left a | Right b
   ```

   As its name suggests, this type is useful when you have a value which can either be of type `a` or type `b`. You might for instance use `Either` for a function that parses integers: The function could have the type `String -> Either String Integer` where a successful parse returns an integer as a `Right` value, whereas an input that cannot be parsed returns an error message as a `Left` value.

   Now to the task at hand: It can sometimes be interesting to determine an implementation for a given type signature. For example, the signature `Either a b -> Either b a` has the following implementation:

   ```
   f :: Either a b -> Either b a
   f (Left x) = Right x
   f (Right x) = Left x
   ```

   Your task is to write total functions that implement the following signatures:

   ```
   g :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
   h :: (a -> Either a b) -> a -> b
   ```

   Your functions may not throw exceptions or be `undefined` and if possible they should terminate on all inputs.

1.3 Abstract Data Types

1. [sheet 12] In this exercise you will model *vectors*, which are essentially resizable arrays. By convention, we will index the cells of our vectors beginning with 0.

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

   ```
   newtype Vector a = ...
   newVector :: Int -> Vector a
   size :: Vector a -> Int
   capacity :: Vector a -> Int
   resize :: Vector a -> Int -> Vector a
   set :: Vector a -> a -> Int -> Maybe (Vector a)
   get :: Vector a -> Int -> Maybe a
   ```

   `newVector n` should return an empty vector of size n. `size` returns the size of the vector, whereas `capacity` returns the number of empty cells in the vector.

   `resize` changes the size of the vector by either adding new empty cells or by truncating the cells with the highest indices.

   `set v x i` should set the cell at index `i` to `x`. If `i` is not a valid index, the function should return `Nothing`.

   `get v i` returns the element in cell `i`. If that cell is empty or if `i` is not a valid index, it should return `Nothing`.

1.4 Type inference

1. [endterm 2014] Why is it often better to write `null` xs instead of xs `== []`?
2. [sheet 11] Use the algorithm from the lecture to determine the most general types of the following definitions:

   (a) `ffoldl = foldl . foldl` (where `foldl :: (a -> b -> a) -> a -> [b] -> a`)

   (b) `f x y = y : map (++x) y`

3. [endterm 2020] Give a brief justification why these expressions do not type check.

   (a) `if f x then x else "error"` (where `f :: (a -> Bool)` and `x :: Int`)

   (b) `1 : 2 : f x` (where `f :: (a -> String)` and `x :: Int`)

2 Homework

2.1 Type Classes

1. [sheet 15] Consider the classes `Semigroup` and `Monoid`:

   ```
   class Semigroup a where
     (<>) :: a -> a -> a

   class Semigroup m => Monoid m where
     mempty :: m
   ```

   We define the type of pairs as follows:

```
data Pair a = Pair a a
```

Your task is to write instances of `Semigroup` and `Monoid` for `Pair`. For `Semigroup`, you have to implement the operation `<>` which should combine two pairs by applying `<>` componentwise. Make sure the operation is associative:

```
Pair a b <> (Pair c d <> Pair e f) =
(Pair a b <> Pair c d) <> Pair e f
```

`Monoid` requires you to give a neutral element `mempty` with respect to `<>`, i.e:

```
Pair a b <> mempty = mempty <> Pair a b = Pair a b
```

2.2 Algebraic Data Types

1. [endterm 2015]

   (a) Define the functions `safeHead :: [a] -> Maybe a` and `safeLast :: [a] -> Maybe a` that, if possible, return the first and last element of a list respectively (otherwise `Nothing`). Do not use any predefined functions.

   (b) Define `safeHead` and `safeLast` again. Use `foldr`, but none of the following techniques: list comprehensions, recursion, pattern matching on lists and predefined functions (except `foldr`).
   Note: `foldr` is defined as follows:

   ```
   foldr :: (a -> b -> b) -> b -> [a] -> b
   foldr f a [] = a
   foldr f a (x:xs) = f x (foldr f a xs)
   ```

   (c) Define the function `select :: Eq a => a -> [(a,b)] -> [b]` that given a value x and a list of pairs (y, z) returns all values z such that $y = x$. Use `map` and `filter`, but none of the following techniques: list comprehensions, recursion and pattern matching on lists.
   Examples:

   ```
   select ’ a ’ [(’b’, 1), (’a’, 3), (’c’, 4), (’a’, 2)] == [3, 2]
   select ’ d ’ [(’b’, 1), (’a’, 3), (’c’, 4), (’a’, 2)] == []
   ```

2. [endterm 2014] `Node` is a datatype representing nodes in a file system:

   ```
   data Node = File String | Dir String [Node]
   ```

   A node is either a *normal file* (`File`) or a *directory* (`Dir`). Nodes have a name (of type `String`). Additionally, directories contain a list of nodes. An entire file system can be represented by a list of nodes:

   ```
   type FileSys = [Node]
   ```

   Example:

```
myFileSys =
  [Dir "Applications"
     [Dir "Aquamacs.app" [File "Info.plist"],
      Dir "Scribus.app" [File "Info.plist"]],
   Dir "Library"
     [Dir "Automator" [],
      Dir "Logs" []],
   Dir "Users"
     [Dir "smith"
        [Dir "bugs" [File "Scratch.hs"],
         File "Scratch.hs"]]]
```

Implement a function `removeFiles :: String -> FileSys -> FileSys` that deletes all normal files with the given name.

3. [sheet 9] An *atom* is either a variable labelled by a string or the value `T` representing truth. A *literal* is a positive or negative atom. Finally, a *formula* is either a literal, a conjunction of formulas, or a disjunction of formulas.

   (a) Define data types for atoms, literals, and formulas. Make your definitions derive instances for `Eq` and `Show`.

   (b) Define values `top :: Literal` and `bottom :: Literal` representing truth and falsity, respectively.

   (c) A *clause* is a disjunction of literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. We define the corresponding types

   ```
   type Clause = [Literal]
   type CNF = [Clause]
   ```

   By convention, an empty clause corresponds to falsity since it is the neutral element of the disjunction operator. Similarly, `[] :: CNF` corresponds to truth - as it is the neutral element of the conjunction operator.

   Define a function `conjToForm :: CNF -> Formula` that transforms a formula given as a list of clauses to a formula as encoded in subtask (a). Examples:

   ```
   conjToForm [] = L top
   conjToForm [[]] = L bottom
   conjToForm [[] , []] = L bottom :&: L bottom
   conjToForm [[Pos $ Var "v1", Neg $ Var "v2"], [Neg $ Var "v1", top]]
     = (L (Pos (Var "v1")) :|: L (Neg (Var "v2"))) :&:
       (L (Neg (Var "v1")) :|: L top)
   ```

   Hint: It might be useful to first define a function of type `Clause -> Formula`.

   (d) Given the type of *valuations* `type Valuation = [(Name,Bool)]` write a function `substConj :: Valuation -> CNF -> CNF` that replaces the variables of a formula by the values as specified in the passed valuation. Examples:

   ```
   substConj [("v", True)] [[Neg $ Var "v"], [Pos $ Var "w"]]
     = [[bottom], [Pos (Var "w")]]
   substConj [("v", False)] [[Neg $ Var "v"], [Pos $ Var "w"]]
     = [[top], [Pos (Var "w")]]
   substConj [("v", True), ("w", False)]
     [[Neg $ Var "v"], [Pos $ Var "w", Neg $ Var "w"]]
     = [[bottom], [bottom, top]]
   ```

(e) Write a function `simpConj :: CNF -> CNF` that simplifies a formula F in CNF in the following way:

   i. For every clause C in F containing `top`, simplify C to `[]top]`.
   ii. For every clause C in F, remove every occurence of `bottom`.
   iii. Remove every clause C in F that has been simplified to `[top]`.
   iv. If F contains a clause C that has been simplified to the empty clause, simplify F to `[[]]`.

   Examples:

```
simpConj [[top], [top, Pos $ Var "v"]] = []
simpConj [[bottom], [Neg $ Var "v"]] = [[]]
simpConj [[Neg $ Var "v"], [bottom, Neg $ Var "v"]]
  = [[Neg (Var "v")], [Neg (Var "v")]]
simpConj [[Pos $ Var "v", bottom], [Pos $ Var "v", top]]
  = [[Pos (Var "v")]]
simpConj [[Neg $ Var "v"], [Neg $ Var "v"],
          [Pos $ Var "w", Neg $ Var "w"]]
  = [[Neg (Var "v")], [Neg (Var "v")], [Pos (Var "w"), Neg (Var "w")]]
```

(f) Write a function `cnf :: Formula -> CNF` that transforms a formula into CNF. Note that if $\phi = \phi_1 \wedge \cdots \wedge \phi_n$ and $\psi = \psi_1 \wedge \cdots \wedge \psi_m$ are in CNF, then the CNF of $\psi \vee \phi$ can be obtained by computing

$$\begin{aligned}
&(\phi_1 \vee \psi_1) \wedge \cdots \wedge (\phi_1 \vee \psi_m) \\
\wedge&(\phi_2 \vee \psi_1) \wedge \cdots \wedge (\phi_2 \vee \psi_m) \\
&\vdots \\
\wedge&(\phi_n \vee \psi_1) \wedge \cdots \wedge (\phi_n \vee \psi_m)
\end{aligned}$$

   Make sure that, for every `cf :: ConjForm`, your functions satisfy the following property: `simpConj cf == simpConj $ cnf $ conjToForm cf`. Example:

```
let (a, b, c, d) = (Pos (Var "a"), Pos (Var "b"),
  Pos (Var "c"), Pos (Var "d"))
cnf $ (L a :&: L b) :|: (L c :&: L d)
  = [[a, c], [a, d], [b, c], [b, d]]
```

4. [1, p. 314] In this exercise we have a closer look at type inference in Haskell. The most fundamental part of type inference algorithms for a type system using parametric polymorphism is the process of *unifying* two types T and U into the most general type V such that both T and U are so-called *instances* of type V. For example, the type `(Int,[Char])` is the result of unifying the types `(a,[Char])` and `(Int,[b])`.

   With *type expressions* we refer to composite types. For example, `(a,[Char])` is a type expression.

   Remember that type variables are universally quantified. An *instance* of a type is given by replacing a type variable or variables by type expressions. All instances of a type form a set describing all possible interpretations of that type.

   A type expression is a *common* instance of two type expressions if it is an instance of each expression. The most general common instance of two expressions is a common instance *mgci* with the property that every other most common instance is an instance of *mgci*.

   The intersection of the sets given by two type expressions is called *unification* of the two, which is the most general common instance of the two type expressions.

   We define the datatype

```
data Type = TypeVar Char | Type String [Type] deriving Eq
```

representing type expressions being either a type variable or a type literal that can have type arguments. For example, the type

```
Tuple a (List Char)
```

is represented as

```
Type "Tuple" [TypeVar 'a', Type "List" [Type "Char" []]]
```

(a) Define an instance of `Show` for `Type` using the same format as in the given example. *Bonus:* print the types `Tuple` and `List` in a more Haskell-like format.

(b) Define a function `unify :: Type -> Type -> Either String Type` that decides whether two type expressions are unifiable. If they are, `unify` should return a most general unifying substitution; if not, it should give some explanation of why the unification fails. You may disregard the case of unifying two type variables.

2.3 Abstract Data Types

1. [sheet 12] Association lists `Eq k => [(k,v)]` are a way of representing maps with keys k and values v. In order to prevent users from creating invalid association lists (e.g. containing muliple values for some key), we want to hide the implementation in a module.

   (a) 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)`.

   (b) 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.

   (c) 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`.

2.4 Type inference

1. [endterm 2013] Determine the most general type of these expressions:

   (a) `filter not`

(b) `[] :  []`
(c) `\f x y -> f (x,y)`
(d) `map (map fst)`

2. [endterm 2013] Give a brief justification why the following expression does not type- check.

```
map head [True, False]
```

3. [1, p. 319] What are the types of

   (a) `curry id`
   (b) `uncurry id`
   (c) `curry (curry id)`
   (d) `uncurry (uncurry id)`
   (e) `uncurry curry`

4. [1, p. 319] Explain why the following expressions do not type-check:

   (a) `curry uncurry`
   (b) `curry curry`

5. [endterm 2020] Determine the most general type of these expressions:

   (a) `foldr (\x y -> y ++ x) []` (where `foldr ::  (a -> b -> b) -> b -> [a] -> b`)
   (b) `(\f g x -> g $ f $ x)`
   (c) `(:[1,2])`
   (d) `map head .  map (\f -> f "hello")`

References

[1] Simon Thompson. *Haskell - the craft of functional programming.* 3rd ed. Pearson, 2011.