Language Engineering - Problem Sheet 3

Dom Hutchinson

November 1, 2018

Question 1 - Simple Expressions
Consider the following simple expression language consisting of just values and additions.

```
data Expr = Val Int
          | Add Expr Expr
```

Question 1.1
Write a recursive funciton *eval* :: *Expr* → *Int* which evaluates these arithmetic expressions.

My Solution 1.1

```
eval :: Expr → Int
eval (Val n) = n
eval (Add x y) = (eval x) + (eval y)
```

Question 1.2

Question 1.2.1
Define the datatype $Fix\ f$ with a single constructor called $In :: f\ (Fix\ f) \to Fix\ f$. Also define $inop :: Fix\ f \to f\ (Fix\ f)$ which unwraps one layer of structure.

My Solution 1.2.1

```
data Fix f = In (f (Fix f))

inop :: Fix f -> f (Fix f)
inop (In x) = x
```

Question 1.2.2
Define a new datatype *ExprF k* which mirrors the constructors of *Expr* except that the parameter *k* replaces recursive occurences of *Expr*.

My Solution 1.2.2

```
data ExprF k = ValF Int
             | AddF k k
```

Question 1.2.3
Let $Expr'$ be a type alias for $Fix\ ExprF$. Write down three values of type $Expr'$.

My Solution 1.2.3

```
ex1 = In(ValF 1)
ex2 = In(ValF 4)
ex3 = In(AddF (In (ValF 1)) (In (ValF 2)))
```

Question 1.2.4
Define the recursive function $fromExpr :: Expr \to Expr'$ that converts values from $Expr$ to $Expr'$.

My Solution 1.2.4

```
fromExpr :: Expr -> (Fix ExprF)
fromExpr (Val n) = In (ValF n)
fromExpr (Add x y) = In(AddF (fromExpr x) (fromExpr y))
```

Question 1.3

Question 1.3.1
Define a function $cata :: Functor\ f \Rightarrow (f\ a \to a) \to Fix\ f \to a$ which reduces the $Fix\ f$ structure into a single value of type a using the provided algebra of type $f\ a \to a$.

My Solution 1.3.1

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg (In x) = alg (fmap (cata alg) x)
```

Question 1.3.2
Give a *Functor* instance for your *ExprF* type.

My Solution 1.3.2

```
instance Functor ExprF
  where
    fmap f (ValF n) = ValF n
    fmap f (AddF x y) = AddF (f x) (f y)
```

Question 1.3.3
Write a function $eval' :: Expr' \to Int$ using the function *cata* and an appropriate algebra.

My Solution 1.3.3

```
eval' :: Fix ExprF-> Int
eval' = cata alg
  where
    alg :: ExprF Int -> Int
    alg (ValF n) = n
    alg (AddF x y) = x+y
```

Question 1.3.4
Discuss the advantages of using *cata* to define the evaluation function for the $Expr'$ type versus the original *eval* for *Expr*.

My Solution 1.3.4
TODO

Question 1.4

Question 1.4.1
Define the function $toExpr :: Expr' \to Expr$ that converts values from $Expr'$ to $Expr$. This must not be a recursive function.

My Solution 1.4.1

```
toExpr :: Fix ExprF -> Expr
toExpr = cata alg
  where
    alg :: ExprF (Expr) -> Expr
    alg (ValF n) = Val n
    alg (AddF x y) = Add x y
```

Question 1.4.2
An isomorphism between two types A and B, written $A \cong B$, exists when there are functions $f : A \to B$ and $g : B \to A$ such that $f \circ g = id$ and $g \circ f = id$. Prove that $Expr \cong Expr'$.

My Solution 1.4.2

TODO

Question 2 - Composing Expressions
It is possible to take the coproduct of functors f and g as given by the following type. This is a type operator that we have declared ot be right associative with precedence 5.

```
data (f :+: g) a = L (f a)
                 | R (g a)
infixr 5 :+:
```

Question 2.1
Give the *Functor* instance for $f : + : g$ under teh assumption that f and g are functors.

My Solution 2.1

```
instance (Functor f, Functor g) => Functor (f:+:g)
  where
    fmap f (L x) = L (fmap f x)
    fmap f (R y) = R (fmap f y)
```

Question 2.2
The datatpe $ExprF$ you defined in the last section can be decomposed into two parts: $ValF\ k$ and $AddF\ k$ which represent the abstract syntax for values and addition respectively.

Question 2.2.1
Give the definitions of both $ValF$ and $AddF$.

My Solution 2.2.1

```
data ValF k = ValF k
data AddF k = AddF k k
```

Question 2.2.2
Give *Functor* instances for both $ValF$ and $AddF$.

My Solution 2.2.2

```
instance Functor ValF
  where
    fmap f (ValF n) = ValF (f n)

instance Functor AddF
  where
    fmap f (AddF x y) = AddF (f x) (f y)
```

Question 2.3
Write a new datatype $SubF\ k$ which represents the abstract syntax for subtraction. Furthermore, write a $Functor$ instance for $SubF$.

My Solution 2.3

```
data SubF k = SubF k k

instance Functor SubF
  where
    fmap f (SubF x y) = SubF (f x) (f y)
```

Question 2.4
Write a function $evalAddSub :: Fix(ValF :+: AddF :+: SubF) \to Int$ using $cata$ which can evaluate arithmetic expressions containing values, additions and subtractions.
Hint - Look at the associativity of the $:+:$ operator.

My Solution 2.4

```
(∇) :: (f a -> a) -> (g a -> a) -> (f :+: g) a -> a
(falg ∇ galg) (L x) = falg x
(falg ∇ galg) (R x) = galg x


val :: ValF Int -> Int
val (ValF x) = x

add :: AddF Int -> Int
add (AddF x y) = x + y

sub :: SubF Int -> Int
sub (SubF x y) = x - y

evalAddSub :: Fix (ValF :+: AddF :+: SubF) -> Int
evalAddSub = cata (val ∇ (add ∇ sub))
```

Question 2.5
Explain why this method of composing datatypes will become more cumbersome as more datatypes are added into the composition.

My Solution 2.5
TODO

Question 3 - Classy Algebras
Consider the new typeclass $Alg\ f\ a$ which contains the operation of an algebra, $alg :: f\ a \to a$, where f is a functor and a is the carrier.
This class requires multiple parameters.

```
class Functor f ⇒ Alg f align
  where
    alg :: f a → a
```

Question 3.1
Give an instance for Alg for $ValF$ with carrier Int.

My Solution 3.1

```
instance Alg ValF Int
  where
   -- alg :: ValF Int -> Int
    alg (ValF x) = x
```

Question 3.2
Give an instance for *Alg* for *AddF* with carrier *Int*.

My Solution 3.2

```
instance Alg AddF Int
  where
    alg (AddF x y) = x + y
```

Question 3.3
Give an instance for *Alg* for *SubF* with carrier *Int*.

My Solution 3.3

```
instance Alg SubF Int
  where
    alg (SubF x y) = x - y
```

Question 3.4
Give the instance for $Alg\ (f :+: g)\ a$, assuming that instances of $Alg\ f\ a$ and $Alg\ g\ a$ exist.
Hint - Use class constraints on your instance.

My Solution 3.4

```
instance (Alg f a, Alg g a) => Alg (f :+: g) a
  where
   -- alg :: (f :+: g) a -> a
    alg (L x) = alg x
    alg (R y) = alg y
```

Question 3.5
Now give a new definition for $evalAddSub :: Fix\ (ValF :+: AddF :+: SubF) \to Int$ using *cata* and an appropriate algebra.
Hint - Your answer should use the algebra from the *Alg* class.

My Solution 3.5

```
evalAddSub' :: Fix (ValF :+: AddF :+: SubF) -> Int
evalAddSub' = cata alg
```

Question 3.6
Define a function $cati :: Alg\ f\ a \Rightarrow Fix\ f \to a$ which performs teh same role as *cata*, but using the *alg* given by $Alg\ g\ a$ instance.

My Solution 3.6

```
cati :: Alg f a => Fix f -> a
cati (In x) = alg (fmap cati x)
```

Question 3.7

Question 3.7.1
Extend the expression language you have built so far by including a new datatype $MulF$.

My Solution 3.7.1

```
data MulF k = MulF k k

instance Functor MulF
  where
    fmap f (MulF x y) = MulF (f x) (f y)

mul :: MulF Int -> Int
mul (MulF x y) = x * y
```

Question 3.7.2
Define an appropriate algebra with carrier Int that performs the multiplication.

My Solution 3.7.2

```
instance Alg MulF Int
  where
    alg (MulF x y) = x * y
```

Question 3.7.3
Redefine $Expr$ to be a synonym for the expression language that includes multiplication.

My Solution 3.7.3

```
type ExprF = Fix (ValF :+: AddF :+: MulF :+: SubF)
```

Question 3.7.4
Use *cati* to define a function $eval :: Expr \to Int$ that evaluates expressions.

My Solution 3.7.4

```
eval :: ExprF -> Int
eval = cati
```

Question 3.8
Without changing any previous code, write a function $depth :: Expr \to Int$ which returns the depth of the deepest node in an expression.
Hint - Use a newtype for Int to provide a specialisation carrier.

My Solution 3.8

TODO