Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Tuesday, March 19th, 2019, Samhällsbyggnad, 8:30 (4hs)

(including example solutions to programming problems)

Alejandro Russo, tel. 0729744968

- The maximum amount of points you can score on the exam: 60 points. The grade for the exam is as follows:

  Chalmers: **3**: 24 - 35 points, **4**: 36 - 47 points, **5**: 48 - 60 points.
  GU: Godkänd 24-47 points, Väl godkänd 48-60 points
  PhD student: 36 points to pass.

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):** Dictionary (Ordlista/ordbok).

  You may bring up to two pages (on one A4 sheet of paper) of pre-written notes — a "summary sheet". These notes may be typed or handwritten. They may be from any source. If this summary sheet is brought to the exam it must also be handed in with the exam (so make a copy if you want to keep it).

- **Notes:**
  - Read through the paper first and plan your time.
  - Answers preferably in English, some assistants might not read Swedish.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of Haskell is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.
  - Hand in the summary sheet (if you brought one) with the exam solutions.
  - As a recommendation, consider spending around 40 minutes per exercise. However, this is only a recommendation.
  - To see your exam: *by appointment (send email to Alejandro Russo)*

Problem 1: (**Monad transformers**)

As described by the lectures, when constructing monads using the state (*StateT*) and the error (*ExceptT*) monad transformers, it is important to determine the order in which you apply them since such decision affects the semantics of your monad. To illustrate this point, we start by showing the following monad:

```
newtype Monad1 a = MkMonad1 (StateT Int (ExceptT String Identity) a)
  deriving (Functor, Applicative, Monad, MonadState Int, MonadError String)
runMonad1 :: Monad1 a → Either String a
runMonad1 (MkMonad1 st) = runIdentity (runExceptT (evalStateT st 0))
```

Monad *Monad1* has been built using the *ExceptT* on the *inside* and then applying *StateT* on top. In contrast, the following monad has *ExceptT* on the *outside*.

```
newtype Monad2 a = MkMonad2 (ExceptT String (StateT Int Identity) a)
  deriving (Functor, Applicative, Monad, MonadState Int, MonadError String)
runMonad2 :: Monad2 a → Either String a
runMonad2 (MkMonad2 er) = runIdentity (evalStateT (runExceptT er) 0)
```

Your task is to write a piece of code that shows the difference of the semantics when being considered as a *Monad1* or *Monad2* computation. That is, you should look for a piece of code t such that the following holds:

$$runMonad1\ t \not\equiv runMonad2\ t$$

Solution:

```
test = runMonad1 (do
    st ← get
    catchError (put (st + 1) ≫ throwError "Error!") (\_ → get))
  ≢
  runMonad2 (do
    st ← get
    catchError (put (st + 1) ≫ throwError "Error!") (\_ → get))
```

(12p)

Problem 2: (**Optimization**)

Consider the following implementation that computes the average on a list of integers.

$average :: [Int] \to Int$
$average\ xs = sum\ xs\ `div`\ length\ xs$

$sum\ [\,] \quad = [\,]$
$sum\ (x : xs) = x + sum\ xs$

$length\ [\,] \quad = 0$
$length\ (x : xs) \quad = 1 + length\ xs$

This function traverses the list *twice*, once to compute the sum and another time to compute the length of it. To reduce the number of passes, we can apply the technique of tupling. This technique consists of obtaining, by equational reasoning, a function that traverse the list *once* while computing two results. In our case, we need to obtain the definition of a function, let's call it $sumlen :: [Int] \to (Int, Int)$, such that the following equation holds.

$$sumlen\ xs \equiv (sum\ xs, length\ xs)$$

Your task is to obtain the definition of *sumlen* by means of equational reasoning so that you can be certain that your definition fulfills the above. You should then implement *average* using *sumlen*.

Solution:

$sumlen\ [\,] \equiv (sum\ [\,], length\ [\,])$
-- by definition of sum.0 and length.0
$\equiv (0, 0)$
$sumlen\ (x : xs) \equiv (sum\ (x : xs), length\ (x : xs))$
-- by definition of sum.1 and length.1
$\equiv (x + sum\ xs, 1 + length\ xs)$
-- by let-definition
$\equiv \textbf{let}\ (s, l) = (sum\ xs, length\ xs)\ \textbf{in}\ (x + s, 1 + l)$
-- by our equation on sumlen on xs
$\equiv \textbf{let}\ (s, l) = sumlen\ xs\ \textbf{in}\ (x + s, 1 + l)$

$sumlen\ [\,] = (0, 0)$
$sumlen\ (x : xs) = \textbf{let}\ (s, l) = sumlen\ xs\ \textbf{in}\ (x + s, 1 + l)$
$average'\ xs = s\ `div`\ l$
$\quad \textbf{where}\ (s, l) = sumlen\ xs$

(12p)

Problem 3: (**Verification**)

We have two known functions to fold over a list with an operator, i.e., to intercalate an operator among the elements of a list.

$$
\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ \oplus\ e\ [\,] \qquad = e \\
&\mathit{foldl}\ \oplus\ e\ (x : xs) = \mathit{foldl}\ \oplus\ (e\ \oplus\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ \oplus\ e\ [\,] = e \\
&\mathit{foldr}\ \oplus\ e\ (x : xs) = x\ \oplus\ (\mathit{foldr}\ \oplus\ e\ xs)
\end{aligned}
$$

Let's see some examples:

$$
\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ [42, 100, 500, 700] \equiv 42 + (100 + (500 + (700 + 0))) \\
&\mathit{foldl}\ (+)\ 0\ [42, 100, 500, 700] \equiv ((((0 + 42) + 100) + 500) + 700)
\end{aligned}
$$

As you see above, the operator $(+)$ has been intercalated among the elements of the list. The difference between *foldr* and *foldl* is where the parentheses are placed. In *foldr*, the parentheses are placed towards the right, while in *foldl* towards the left—therefore the names! In the example above, both functions arrive at the same result since $(+)$ is associative and 0 is the neutral element. More generally, we have the following formal result:

Theorem Given an associative operator $\oplus$ with neutral element e, it holds that $\mathit{foldr}\ \oplus\ e\ xs \equiv \mathit{foldl}\ \oplus\ e\ xs$.

Your task is to prove the theorem.

Solution:

We need to first prove the following lemma $\mathit{foldl}\ \oplus\ y\ ys \equiv y\ \oplus\ \mathit{foldl}\ \oplus\ e\ ys$ by induction on the length of ys.

$$
\begin{aligned}
&\mathit{foldl}\ \oplus\ y\ [\,] \equiv && \text{-- by definition of foldl.0} \\
&y \equiv && \text{-- by neutral e} \\
&y\ \oplus\ e \equiv && \text{-- by definition of foldl.0} \\
&y\ \oplus\ (\mathit{foldl}\ \oplus\ e\ [\,])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldl}\ \oplus\ y\ (x : xs) \equiv && \text{-- def. foldl.1} \\
&\mathit{foldl}\ \oplus\ (y\ \oplus\ x)\ xs \equiv && \text{-- by IH} \\
&(y\ \oplus\ x)\ \oplus\ (\mathit{foldl}\ \oplus\ e\ xs) \equiv && \text{-- by assoc.} \\
&y\ \oplus\ (x\ \oplus\ (\mathit{foldl}\ \oplus\ e\ xs)) \equiv && \text{-- by IH} \\
&y\ \oplus\ (\mathit{foldl}\ \oplus\ x\ xs) \equiv && \text{-- by neutral} \\
&y\ \oplus\ (\mathit{foldl}\ \oplus\ (e\ \oplus\ x)\ xs) \equiv && \text{-- by foldl.1} \\
&y\ \oplus\ (\mathit{foldl}\ \oplus\ e\ (x : xs))
\end{aligned}
$$

With this lemma in place, we then proceed to prove the theorem.

$$
\begin{aligned}
&\mathit{foldr}\ \oplus\ e\ [\,] \equiv && \text{-- by foldr.0} \\
&e \equiv && \text{-- by foldl.0} \\
&\mathit{foldl}\ \oplus\ e\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldr}\ \oplus\ e\ (x : xs) \equiv && \text{-- by foldr.1} \\
&x\ \oplus\ (\mathit{foldr}\ \oplus\ e\ xs) \equiv && \text{-- by IH}
\end{aligned}
$$

$$
\begin{array}{ll}
x \oplus (\mathit{foldl} \oplus e\ xs) \equiv & \text{-- by lemma} \\
\mathit{foldl} \oplus x\ xs \equiv & \text{-- by neutral} \\
\mathit{foldl} \oplus (e \oplus x)\ xs \equiv & \text{-- def. foldl.1} \\
\mathit{foldl} \oplus e\ (x : xs) &
\end{array}
$$

(12p)

Problem 4: (Type level programming)

In this exercise, we will implement heterogeneous lists in Haskell, i.e., lists where the elements can have different types!

We start by assuming that we have the *DataKinds* extension enable, which gives us type-level lists, i.e., we have the *types* [ ], 42 : [ ], etc. Now, we will use the power of GADTs to introduce heterogeneous lists.

```
data HList xs where
  HNil :: ...
  (:::)  :: ...
```

a) Your task is to complete the type signatures for *HNil* and (:::). Your implementation should be able to implement the following examples.

```
ex1 :: HList [Char, Integer, Double, [Integer]]
ex1 = 'a' ::: 42 ::: 1.0 ::: [42] ::: HNil
ex2 :: HList [[Double], Char]
ex2 = [1.0] ::: 'b' ::: HNil
```

Solution:

```
data HList xs where
  HNil :: HList []
  (:::) :: a → HList as → HList (a : as)
```

(6p)

b) We can now build heterogeneous lists, but we cannot show them. In order to show them on the screen, we need to have instances of *Show* for *HList xs*. Your task is to provide instances for *Show*.

```
> ex1
'a' ::: 42 ::: 1.0 ::: [42] ::: HNil
> ex2
[1.0] ::: 'b' ::: HNil
```

Solution:

```
instance Show (HList []) where
  show HNil = "HNil"
instance (Show (HList as), Show a)
  ⇒ Show (HList (a : as)) where
  show (a ::: rest) =
    show a ++ " ::: " ++ show rest
infixr 6 :::
```

(6p)

Problem 5: (**Monads transformers**)

In the lectures and in the last lab this year, we show how information-flow control (IFC) is a promising technology to guarantee confidentiality of data when manipulated by untrusted code (i.e., code written by someone else) as well as buggy code (i.e., code written perhaps by ourselves or a colleague).

To build secure programs which do not leak secrets, we studied a small EDSL in Haskell with two core concepts: labeled values and secure computations. The following code is the core implementation of the MAC library given in the lectures:

```
{-Points in the security lattice -}
data L
data H

{-Order relationship of the lattice -}
class l ⊑ l' where

instance L ⊑ L where
instance L ⊑ H where
instance H ⊑ H where

{-Labeled values -}
newtype Labeled l a = Labeled a

{-Secure computation with underlying IO actions -}
newtype MAC l a = MkMAC (IO a)

{-Functor, Applicative, and Monad instances -}
instance Functor (MAC l) where
  fmap f (MkMAC io) = MkMAC (fmap f io)

instance Applicative (MAC l) where
  pure = MkMAC ∘ return
  (<*>) (MkMAC f) (MkMAC a) = MkMAC (f <*> a)

instance Monad (MAC l) where
  return = pure
  MkMAC m >>= k = MkMAC (m >>= runMAC ∘ k)

{-Primitive combinators -}
runMAC :: MAC l a → IO a
runMAC (MkMAC m) = m

label :: l ⊑ l' ⇒ a → MAC l (Labeled l' a)
label v = return (Labeled v)

unlabel :: l ⊑ l' ⇒ Labeled l a → MAC l' a
unlabel (Labeled v) = return v

joinMAC :: l ⊑ l' ⇒ MAC l' a → MAC l (Labeled l' a)
joinMAC m = MkMAC (runMAC m) >>= label
```

Above, the constructors *MkMAC* and *Labeled* are never exported so that users of the DSL cannot break its abstraction. In this exercise, you will need to create a monad transformer for *MAC*, which we call *MACT*.

```
data MACT m l a
```

The idea is that when applying $MACT$ to a monad m, then we obtain a monad capable to perform the effects of m as well as keeping sensitive information secret. For instance, $MACT\ l\ (State\ s)\ a$ is a secure state monad with state s.

a) Your task is to give the instances for *Functor*, *Applicative* and *Monad* for $MACT\ l\ m$ for your implementation.

Solution:

```
newtype MACT m l a = MACT {runMACT :: m a}
instance Monad m => Functor (MACT m l) where
  fmap f (MACT m) = MACT (fmap f m)
instance Monad m => Applicative (MACT m l) where
  pure = MACT ∘ return
  (<*>) (MACT f) (MACT a) = MACT (f <*> a)
instance Monad m => Monad (MACT m l) where
  return = pure
  MACT m >>= k = MACT (m >>= runMACT ∘ k)
```

(8p)

b) Since $MACT\ l\ m$ might create many security monads, e.g., $MACT\ l\ Identity\ a$, $MACT\ l\ (State\ s)\ a$, $MACT\ l\ IO\ a$, that means that there would be many implementation of *label*, *unlabel*, and *join*. In this light, we need to overload such operators and we do so in the following type class.

```
class MACMonad m where
  label'   :: LessEq l l' => a -> m l (Labeled l' a)
  unlabel' :: LessEq l l' => Labeled l a -> m l' a
  join'    :: LessEq l l' => m l' a -> m l (Labeled l' a)
```

Observe that we have "primed" the operators so that there is no name clashing with the code in part a), i.e., you see *label′* rather than *label*.

Your task is to provide the instance of $MACMonad\ m$ when m is obtained by applying your monad transformer $MACT\ l$ to the an arbitrary monad m. In other words, you should provide the code for the following instance:

```
instance Monad m => MACMonad (MACT m) where
```

Solution:

```
instance Monad m => MACMonad (MACT m) where
  label' a = MACT (return (Labeled a))
  unlabel' (Labeled a) = MACT (return a)
  join' m = (MACT (runMACT m)) >>= label'
```

(4p)