**Chalmers** | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

# Advanced Functional Programming TDA342/DIT260

Tuesday, March 15, 2016, Hörsalsvägen (yellow brick building), 8:30-12:30.

(including example solutions to programming problems)

Alejandro Russo, tel. 031 772 6156

- 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 1h 20 minutes per exercise. However, this is only a recommendation.
  - To see your exam: *by appointment (send email to Alejandro Russo)*

| | |
|---|---|
| FUNCTOR TYPE-CLASS<br>**class** *Functor* $c$ **where** $fmap :: (a \to b) \to c\ a \to c\ b$ | IDENTITY<br>$fmap\ id \equiv id$ **where** $id = \lambda x \to x$ |

MAP FUSION

$fmap\ (f \circ g) \equiv fmap\ f \circ fmap\ g$

Figure 1: Functors

**Problem 1:** (**Functors**) As its name implies, a binary tree is a tree with a two-way branching structure, i.e., a left and a right sub tree. In Haskell, such trees can be defined as follows.

```
data Tree a where
  Leaf :: a -> Tree a
  Node :: Tree a -> Tree a -> Tree a
```

**a)** Show that *Tree a* is a functor. For that, you should *provide* an instance for the *Functor* type-class and *prove* that *fmap* for *finite trees*, i.e., $fmap :: (a \to b) \to Tree\ a \to Tree\ b$, fulfills the laws for functors – see Figure 1.

```
instance Functor Tree where
  fmap f (Leaf a)      = Leaf (f a)
  fmap f (Node t1 t2) = Node (fmap f t1) (fmap f t2)
```

**Important**: Assume that $f$ and $g$ are total, i.e., they do not raise any errors or loop indefinitely when applied to an argument. If your proof is by induction, you should indicate *induction on what* (e.g., in the length of the list). Justify every step in your proof.

*(8p)*

Proofs by induction on the height of the tree

```
-- Identity law
-- Base case
fmap id (Leaf a) ≡
--  by definition fmap.0
Leaf (id a)      ≡
--  by definition of id
Leaf a           ≡
--  by definition of id
id (Leaf a)
-- Inductive case
fmap id (Node l r)             ≡
--  by definition of fmap.1
Node (fmap id l) (fmap id r)  ≡
--  by I.H.
Node (id l) (id r)            ≡
```

-- by definition of id
$Node\ l\ r \equiv$
-- by definition of id
$id\ (Node\ l\ r)$

-- Map fusion

-- Base case
$fmap\ (f \circ g)\ (Leaf\ a) \equiv$
-- by definition fmap.0
$Leaf\ ((f \circ g)\ a) \equiv$
-- by definition of .
$Leaf\ (f\ (g\ a)) \equiv$
-- by definition of fmap.0
$fmap\ f\ (Leaf\ (g\ a)) \equiv$
-- by definition of fmap.0
$fmap\ f\ (fmap\ g\ (Leaf\ a))$

-- Inductive case
$fmap\ (f \circ g)\ (Node\ l\ r) \equiv$
-- by definition of fmap.1
$Node\ (fmap\ (f \circ g)\ l)\ (fmap\ (f \circ g)\ r) \equiv$
-- by I.H.
$Node\ (fmap\ f\ (fmap\ g\ l))\ (fmap\ f\ (fmap\ g\ r)) \equiv$
-- by definition of fmap.1
$fmap\ f\ (Node\ (fmap\ g\ l)\ (fmap\ g\ r)) \equiv$
-- by definition of fmap.1
$fmap\ f\ (fmap\ g\ (Node\ l\ r))$

**b)** As with lists, it is also useful to "fold" over trees. Given a tree $t$ with elements $e_1$, $e_2$, ..., $e_n$ and an operator $\oplus$, folding over the tree $t$ with operator $\oplus$ intuitively means to *intercalate* the operator among the elements of the tree, i.e., $e_1 \oplus e_2 \oplus e_3 \oplus \ldots \oplus e_n$. For simplicity, we assume that the operator $\oplus$ is always associative. We call the function implementing folding over trees *foldT*.

$$foldT :: (a \to a \to a) \to Tree\ a \to a$$

By using *foldT*, we can now express a bunch of useful functions on trees.

$P_1$
$height\_tree = foldT\ (\lambda l\ r \to max\ l\ r + 1) \circ fmap\ (const\ 0)$

$P_2$
$sum\_tree = foldT\ (+)$

$P_3$
$leaves = foldT\ (+\!\!+) \circ fmap\ (\lambda x \to [x])$

Program $P_1$ computes the height of a tree. Program $P_2$ sums all the numbers in a tree. Program $P_3$ extracts all the elements of a tree.

Your task is to implement *foldT*. *(4p)*

$$
\begin{aligned}
&foldT :: (a \to a \to a) \to Tree\ a \to a\\
&foldT\ op\ (Leaf\ a) \quad = a\\
&foldT\ op\ (Node\ l\ r) = (foldT\ op\ l)\ \text{`}op\text{`}\ (foldT\ op\ r)
\end{aligned}
$$

**c)** There is a relation between mapping functions over trees' leaves and lists. More specifically, we have the following equation for finite and well-defined trees.

$$map\ f \circ leaves \equiv leaves \circ fmap\ f$$

It is the same to first extract the leaves and then map the function (left-hand side), as it is to map the function first and then extracting the leaves (right-hand side).

Your task is to prove that the equation holds.

**You can assume the following properties and definition for this exercise and the rest of the exam!**

| (.) | Assoc. (.) | (id left) | (id right) | (eta) |
|---|---|---|---|---|
| $(f \circ g)\ x = f\ (g\ x)$ | $(f \circ g) \circ z = f \circ (g \circ z)$ | $id \circ f = f$ | $f \circ id = f$ | $\lambda x \to f\ x \equiv f$ |

| (cons.0) | ((++).0) | ((++).1) |
|---|---|---|
| $x : [\,] = [x]$ | $[\,] +\!\!+\ ys = ys$ | $(x : xs) +\!\!+\ ys = x : (xs +\!\!+\ ys)$ |

| (Assoc. (++)) | (map.0) | (map.1) |
|---|---|---|
| $xs +\!\!+\ (ys +\!\!+\ zs) \equiv (xs +\!\!+\ ys) +\!\!+\ zs$ | $map\ f\ [\,] = [\,]$ | $map\ f\ (x : xs) = f\ x : map\ f\ xs$ |

You cannot assume any property that relates $(+\!\!+)$, *map*, and *fmap* – if you need such properties, you should prove them too! *(8p)*

-- Auxiliary lemma

$map\ f\ (xs +\!\!+\ ys) \equiv map\ f\ xs +\!\!+\ map\ f\ ys$

-- Proof by induction on the length of xs

-- Base case

$map\ f\ ([\,] +\!\!+\ ys) \equiv$

-- (++).0

$map\ f\ ys \equiv$

-- (++).0

$[\,] +\!\!+\ map\ f\ ys \equiv$

-- map.0

$map\ f\ [\,] +\!\!+\ map\ f\ ys$

-- Inductive case

$map\ f\ ((x : xs) +\!\!+\ ys) \equiv$

-- map.1

$f\ x : map\ f\ (xs +\!\!+\ ys) \equiv$

-- I.H.

$f\ x : (map\ f\ xs +\!\!+\ map\ f\ ys) \equiv$

-- (++).1

$(f\ x : map\ f\ xs) +\!\!+\ map\ f\ ys \equiv$
-- map.1
$map\ f\ (x : xs) +\!\!+\ map\ f\ ys$

-- Proof by induction on the height of trees
$map\ f \circ leaves \equiv leaves \circ fmap\ f$

-- Base case
$map\ f\ (leaves\ (Leaf\ a)) \equiv$
-- Def. leaves
$map\ f\ (foldT\ (+\!\!+)\ (fmap\ (\lambda x \to [x])\ (Leaf\ a))) \equiv$
-- Def. fmap on Leaf
$map\ f\ (foldT\ (+\!\!+)\ (Leaf\ [a])) \equiv$
-- Def. foldT
$map\ f\ [a] \equiv$
-- Def (:)
$map\ f\ (a : [\,]) \equiv$
-- Def map.1
$f\ a : map\ f\ [\,] \equiv$
-- Def. map.0
$f\ a : [\,] \equiv$
-- Def (:)
$[f\ a] \equiv$
-- Def. leaves
$leaves\ (Leaf\ (f\ a)) \equiv$
-- Def. fmap
$leaves\ (fmap\ f\ (Leaf\ a))$

-- Inductive case
$map\ f\ (leaves\ (Node\ l\ r)) \equiv$
-- Def. leaves
$map\ f\ (foldT\ (+\!\!+)\ (Node\ l\ r)) \equiv$
-- Def. foldT
$map\ f\ ((foldT\ (+\!\!+)\ l) +\!\!+\ (foldT\ (+\!\!+)\ r)) \equiv$
-- Auxiliary lemma
$map\ f\ (foldT\ (+\!\!+)\ l) +\!\!+\ map\ f\ (foldT\ (+\!\!+)\ r) \equiv$
-- Def. leaves
$map\ f\ (leaves\ l) +\!\!+\ map\ f\ (leaves\ r) \equiv$
-- IH
$leaves\ (fmap\ f\ l) +\!\!+\ leaves\ (fmap\ f\ r) \equiv$
-- Def. leaves
$foldT\ (+\!\!+)\ (fmap\ f\ l) +\!\!+\ foldT\ (+\!\!+)\ (fmap\ f\ r) \equiv$
-- Def. foldT
$foldT\ (+\!\!+)\ (Node\ (fmap\ f\ l)\ (fmap\ f\ r)) \equiv$
-- Def. leaves
$leaves\ (Node\ (fmap\ f\ l)\ (fmap\ f\ r)) \equiv$
-- Def. fmap

$leaves\ (fmap\ f\ (Node\ l\ r))$

$$\begin{array}{lll}
\textbf{class } \mathit{Monad}\ m\ \textbf{where} & \textsc{Left Identity} & \textsc{Right Identity} \\
\quad \mathit{return} :: a \to m\ a & \mathit{return}\ x \gg\!= f \equiv f\ x & m \gg\!= \mathit{return} \equiv m \\
\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b & &
\end{array}$$

$$\begin{array}{l}
\textsc{Associativity } (x \text{ does not appear in } k_1 \text{ and } k_2) \\
(m \gg\!= k_1) \gg\!= k_2 \equiv m \gg\!= (\lambda x \to k_1\ x \gg\!= k_2)
\end{array}$$

Figure 2: Monads

**Problem 2.** (**Monads**) During the lectures we said that a data type $m$ is a monad if we can define the primitives *return* and $(\gg\!=)$, and that $m$ fulfills the monadic laws – see Figure 2. There is, however, an alternative interface for monads described as follows.

$$\begin{array}{lll}
\textbf{class } \mathit{MonadAlternative}\ m\ \textbf{where} & \textsc{Identity} & \textsc{Map fusion} \\
\quad \mathit{return}' :: a \to m\ a & \mathit{fmap}'\ \mathit{id}\ m \equiv m & \mathit{fmap}'\ (f \circ g) \equiv \mathit{fmap}'\ f \circ \mathit{fmap}'\ g \\
\quad \mathit{join}\ \ :: m\ (m\ a) \to m\ a & & \\
\quad \mathit{fmap}'\ :: (a \to b) \to m\ a \to m\ b & &
\end{array}$$

$$\begin{array}{lll}
A_1 & A_2 & A_3 \\
\mathit{fmap}'\ f \circ \mathit{return}' \equiv \mathit{return}' \circ f & \mathit{join} \circ \mathit{fmap}'\ \mathit{return}' \equiv \mathit{id} & \mathit{join} \circ \mathit{return}' \equiv \mathit{id}
\end{array}$$

$$\begin{array}{ll}
A_4 & A_5 \\
\mathit{join} \circ \mathit{fmap}'\ \mathit{join} \equiv \mathit{join} \circ \mathit{join} & \mathit{join} \circ \mathit{fmap}'\ (\mathit{fmap}'\ f) \equiv \mathit{fmap}'\ f \circ \mathit{join}
\end{array}$$

This interface requires $m$ to be a functor and introduces an operation called *join*. Furthermore, *return*′, *join*, and *fmap*′ are required to obey various different laws.

**a)** Your task consists of showing that the alternative interface is enough to implement *return* and $(\gg\!=)$. In other words, if you define *return*′, *fmap*′, and *join* for certain data type $m$, then you can show that $m$ is an instance of the type-class *Monad* in Haskell. You should provide the following type-class instance:

$$\begin{array}{l}
\textbf{instance } \mathit{MonadAlternative}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} = \ldots \\
\quad (\gg\!=)\ = \ldots
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{MonadAlternative}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return}\ \ = \mathit{return}' \\
\quad m \gg\!= k = \mathit{join}\ (\mathit{fmap}'\ k\ m)
\end{array}$$

*(6p)*

**b)** Assuming the laws for the alternative monadic interface, you should show that the implementation that you gave in the previous question is indeed a monad in the traditional sense, i.e. it fulfills the laws from Figure 2. *(14p)*

-- Left identity

$return\ x \gg\!= f \equiv$

-- Def. return

$join\ (fmap'\ f\ (return\ x)) \equiv$

-- Def. return

$join\ (fmap'\ f\ (return'\ x)) \equiv$

-- Def. of (.)

$join\ ((fmap'\ f \circ return')\ x) \equiv$

-- A1

$join\ ((return' \circ f)\ x) \equiv$

-- Def (.)

$(join \circ return' \circ f)\ x \equiv$

-- A3

$(id \circ f)\ x \equiv$

-- Def. id

$f\ x$

-- Right identity

$m \gg\!= return \equiv$

-- Def. bind

$join\ (fmap'\ return\ m) \equiv$

-- Def. return

$join\ (fmap'\ return'\ m) \equiv$

-- Def. (.)

$(join \circ fmap'\ return')\ m \equiv$

-- A2

$id\ m \equiv$

-- Def. id

$m$

-- Associativity

$m \gg\!= (\lambda x \to k_1\ x \gg\!= k_2) \equiv$

-- Def. bind

$join\ (fmap'\ (\lambda x \to k_1\ x \gg\!= k_2)\ m) \equiv$

-- Def. bind

$join\ (fmap'\ (\lambda x \to join\ (fmap'\ k_2\ (k_1\ x)))\ m) \equiv$

-- Def. (.)

$join\ (fmap'\ (\lambda x \to (join \circ fmap'\ k_2 \circ k_1)\ x)\ m) \equiv$

-- Eta-contraction

$join\ (fmap'\ (join \circ fmap'\ k_2 \circ k_1)\ m) \equiv$

-- Map fusion

$join\ (fmap'\ join\ (fmap'\ (fmap'\ k_2 \circ k_1)\ m)) \equiv$

-- Map fusion

$join\ (fmap'\ join\ (fmap'\ (fmap'\ k_2)\ (fmap'\ k_1\ m))) \equiv$

-- Def (.)

$$
\begin{aligned}
&(join \circ fmap'\ join)\ (fmap'\ (fmap'\ k_2)\ (fmap'\ k_1\ m))) \equiv \\
&\quad \text{-- A4} \\
&(join \circ join)\ (fmap'\ (fmap'\ k_2)\ (fmap'\ k_1\ m))) \equiv \\
&\quad \text{-- Def (.)} \\
&join\ (join\ (fmap'\ (fmap'\ k_2)\ (fmap'\ k_1\ m))) \quad \equiv \\
&\quad \text{-- Def (.)} \\
&join\ ((join \circ fmap'\ (fmap'\ k_2))\ (fmap'\ k_1\ m)) \quad \equiv \\
&\quad \text{-- A5} \\
&join\ ((fmap'\ k_2 \circ join)\ (fmap'\ k_1\ m)) \qquad \equiv \\
&\quad \text{-- Def (.)} \\
&join\ (fmap'\ k_2\ (join\ (fmap'\ k_1\ m))) \qquad \equiv \\
&\quad \text{-- Def. bind} \\
&(join\ (fmap'\ k_1\ m)) \gg= k_2 \qquad \equiv \\
&\quad \text{-- Def. bind} \\
&(m \gg= k_1) \gg= k_2
\end{aligned}
$$

**Problem 3:** (**EDSL**) *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. In IFC, data gets classified either as *public* (low) or *secret* (high), where public information can flow into secret entities but not vice versa. We encode the sensitivity of data as abstract data types, and the allowed flows of information in the type-class *CanFlowTo* – see Figure 3.

To build secure programs which do not leak secrets, we build a small EDSL in Haskell with two core concepts: *labeled values* and *secure computations*. Labeled values are simply data tagged with a security level indicating its sensitivity. For example, a weather report is a public piece of data, so we can model it as a public labeled string *weather_report* :: *Labeled L String*. Similarly, a credit card number is sensitive, so we model it as a secret integer *cc_number* :: *Labeled H Integer*.

```
  -- Security level for public data
data L
  -- Security level for secret data
data H
  -- allowed flows of information
class l `CanFlowTo` l' where
  -- Public data can flow into public entities
instance L `CanFlowTo` L where
  -- Public data can flow into secret entities
instance L `CanFlowTo` H where
  -- Secret data can flow into secret entities
instance H `CanFlowTo` H where
```

Figure 3: Allowed flows of information

A secure computation is an entity of type *MAC l a*, which denotes a computation that handles data at sensitivity level *l* and produces a result (of type *a*) of this level. In order to remain secure, secure computations can only observe data that "can flow to" the computation (see primitive *unlabel* below), and can only create labeled values provided that information from the computation "can flow to" the newly created labeled value (see primitive *label* below). We describe the API for the EDSL in Figure 4, and provide a *deep-embedded* implementation for the API in Figure 5.

**a)** Your task is to take the implementation in Figure 5 and obtain an "intermediate embedding" by removing *Bind* from the *MAC l a* data type. As a result, *runMAC* will no longer run *Bind*; instead, the defintion of (>>=) will change. After your modifications, it is important to show that you can faithfully implement the *whole* EDSL API.

**Important**: If you alter the definition of *MAC l a*, or any other function in the deep-embedded implementation, you need to show that your modifications are correct by deriving them.

**Help:** You can assume that *runMAC* $(m \gg\!= f) \equiv$ *runMAC* $m \gg\!=$ *runMAC* $\circ f$ *(12p)*

```
data MAC l a where
  Label    :: (l `CanFlowTo` l') => Labeled l' a -> MAC l (Labeled l' a)
  Unlabel  :: (l' `CanFlowTo` l) => Labeled l' a -> MAC l a
  JoinBind :: (l `CanFlowTo` l') => MAC l' a
                                 -> ((Labeled l' a) -> MAC l b)
                                 -> MAC l b
  Return   :: a -> MAC l a
```

```
-- Types
newtype Labeled l a
data    MAC l a

-- Labeled values
label    :: (l `CanFlowTo` l') ⇒ a → MAC l (Labeled l' a)
unlabel  :: (l' `CanFlowTo` l) ⇒ Labeled l' a → MAC l a

-- MAC monad
return   :: a → MAC l a
(>>=)    :: MAC l a → (a → MAC l b) → MAC l b
joinMAC  :: (l `CanFlowTo` l') ⇒ MAC l' a → MAC l (Labeled l' a)

-- Run function
runMAC   :: MAC l a → IO a
```

Figure 4: EDSL API

```
-- Types
newtype Labeled l a = MkLabeled a

data MAC l a where
  Label   :: (l `CanFlowTo` l') ⇒ Labeled l' a → MAC l (Labeled l' a)
  Unlabel :: (l' `CanFlowTo` l) ⇒ Labeled l' a → MAC l a
  Join    :: (l `CanFlowTo` l') ⇒ MAC l' a    → MAC l (Labeled l' a)
  Return  :: a → MAC l a
  Bind    :: MAC l a → (a → MAC l b) → MAC l b

-- Labeled values
label    = Label ∘ MkLabeled
unlabel  = Unlabel

-- MAC operations
joinMAC = Join

instance Monad (MAC l) where
  return = Return
  (>>=)  = Bind

-- Run function
runMAC (Label lv)                = return lv
runMAC (Unlabel (MkLabeled v))   = return v
runMAC (Join mac_a)              = runMAC mac_a >>= return ∘ MkLabeled
runMAC (Return a)                = return a
runMAC (Bind mac f)              = runMAC mac >>= runMAC ∘ f
```

Figure 5: Deep-embedded implemention

```
  -- joinMAC
joinMAC mac_h = JoinBind mac_h Return

  -- Implementing bind
instance Monad (MAC l) where
  return = Return
  Label lv                ≫= f = f lv
  Unlabel (MkLabeled v)   ≫= f = f v
  JoinBind mac_h k        ≫= f = JoinBind mac_h (λlv → k lv ≫= f)
  Return x                ≫= f = f x

  -- Derivation for JoinBind
  JoinBind mac_h k ≫= f
  -- definition of JoinBind
  (Join mac_h ≫= k) ≫= f
  -- associativity of bind
  Join mac_h ≫= (λlv → k lv ≫= f)
  -- definition of JoinBind
  JoinBind mac_h (λv → k lv ≫= f)

runMAC (Label lv)               = return lv
runMAC (Unlabel (MkLabeled v))  = return v
runMAC (Return a)               = return a
runMAC (JoinBind mac_h k)       = runMAC mac_h ≫= runMAC ∘ k ∘ MkLabeled

  -- Derivation for runMAC for JoinBind
  runMAC (JoinBind mac_h k)
  -- definition of JoinBind
  runMAC (Join mac_h ≫= k)
  -- property of runMAC and bind
  runMAC (Join mac_h) ≫= runMAC ∘ k
  -- definition runMAC for Join from before
  (runMAC mac_h ≫= return ∘ MkLabeled) ≫= runMAC ∘ k
  -- associativity law for monads
  runMAC mac_h ≫= (λx → (return ∘ MkLabeled) x ≫= runMAC ∘ k)
  -- Definition of . and application
  runMAC mac_h ≫= (λx → return (MkLabeled x) ≫= runMAC ∘ k)
  -- Left identity
  runMAC mac_h ≫= (λx → (runMAC ∘ k) (MkLabeled x))
  -- definition of (.)
  runMAC mac_h ≫= (λx → (runMAC ∘ k ∘ MkLabeled) x)
  -- eta-contraction
  runMAC mac_h ≫= runMAC ∘ k ∘ MkLabeled
```

**b)** We would like to add the function *output* to the EDSL in order to print out messages. Ideally, we will have two output channels, one for public data and one for secret values. However, for simplicity, we assume that we have only one output channel: the screen. To mimic having two output channels, however, we will pre-append some text to indicate on which channel data is being sent. See the functions *add_location* and *print_cc* below.

```
-- outputting in a public channel
add_location :: Labeled L String → MAC L ()
add_location lstr = do
  str ← unlabel lstr
  msg ← label (str ++ "Gothenburg")
          :: MAC L (Labeled L String)
  output msg
```

```
-- outputting in a secret channel
print_cc :: Labeled H Int → MAC H ()
print_cc lcc = do
  number ← unlabel lcc
  msg    ← label ("CC number "
                  ++ show number)
             :: MAC H (Labeled H String)
  output msg
```

If we call *add_location* with a weather report, then it prints out a message in the public channel.

```
> let weather = MkLabeled "Sunny, 31 degrees, " :: Labeled L String
  in runMAC (add_location weather)
public channel : Sunny, 31 degrees, Gothenburg
```

By contrast, if we call *print_cc* with a credit card number, then it sends the credit card digits to the secret channel.

```
> let cc_number = MkLabeled 1234 :: Labeled H Int
  in runMAC (print_cc cc_number)
private channel : CC number 1234
```

Observe that the implementation of *output* depends on the type of the labeled value taken as argument, i.e. *output* is overloaded. Your task is to extend the definitions of *MAC l a*, (>>=), and *runMAC* to include the primitive *output* in the EDSL. *(8p)*

```
class TermLevel l where
  term :: Labeled l a → Level
data Level = Public | Secret
instance TermLevel L where
  term _ = Public
instance TermLevel H where
  term _ = Secret
data MAC l a where
    ...
  Output :: TermLevel l ⇒ Labeled l String → MAC l ()
instance Monad (MAC l) where
```

```
  ...
  Output lv >>= f = f ()
runMAC (Output lv@(MkLabeled msg)) =
  case term lv of
    Public -> putStrLn "public channel:" >> putStrLn msg
    Secret -> putStrLn "secret channel:" >> putStrLn msg
```