Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Tuesday 14th March, 2017, Samhällsbyggnad, 8:30.

Alejandro Russo (Anton Ekblad, tel. 0707 579 070)

- 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 for exercise 1, 1.20h for exercise 2, and 2hs for exercise 3. However, this is only a recommendation.
  - To see your exam: *by appointment (send email to Alejandro Russo)*

Problem 1: (Applicative Functors)

In the lectures, we saw an example of an applicative functor which was not a monad. The example consisted on the data type definition:

$$\textbf{data}\ \mathit{Phantom}\ o\ a = \mathit{Phantom}\ o$$

It is called *Phantom* since it contains no value of type a—it is like an empty body, a spirit, a phantom.

We saw that we can define the instances *Functor* and *Applicative* as follows.

```
instance Functor (Phantom o) where
  fmap _ (Phantom o) = Phantom o
instance Monoid o ⇒ Applicative (Phantom o) where
  pure _                         = Phantom 1
  Phantom o₁ <*> Phantom o₂ = Phantom (o₁ · o₂)
```

In these definitions, we assume a monoid structure for elements of type o, i.e. it contains an identity element 1 and a associative binary operation $(\cdot)$.

In the lectures, we showed that when o is of type *Int*, any implementation of bind, i.e.

$$(\gg\!=) :: \mathit{Phantom}\ \mathit{Int}\ a \to (a \to \mathit{Phantom}\ \mathit{Int}\ b) \to \mathit{Phantom}\ \mathit{Int}\ b$$

violates the left identity law.

i) (**Task**) Come up with a type o' and an implementation of **instance** *Monad* (*Phantom* o'), where *Phantom* o' is indeed a monad, i.e. it respects the monadic laws (see Figure 4). *(4p)*

ii) The *composition* of two functors f and g is defined by the following data type:

```
data Comp c d a = Comp (c (d a))
instance (Functor c, Functor d) ⇒ Functor (Comp c d) where
  fmap f (Comp cda) = Comp (fmap (fmap f) cda)
```

(**Task**) Show that *Comp f g a* is also a functor, so it fulfills the *identity* and *map fusion* laws (see Figure 5). In other words, you will show that the composition of functors results in a functor. *(8p)*

iii) (**Task**) Applicatives are closed under functor composition, too! Define the applicative instance for the composition of two applicatives.

```
instance (Applicative f, Applicative g) ⇒ Applicative (Comp f g) where
  ...
```

Show that your definitions of *pure* and (<*>) satisfy the applicative laws (see Figure 6).

(8p)

Problem 2: (**Type families**)

i) Consider the following EDSL, which lets users perform basic arithmetic without having to worry about dividing by zero:

```
data Exp a where
  Int  ::            Int            → Exp Int
  Doub ::            Double         → Exp Double
  Div  :: Divide a ⇒ Exp a → Exp a → Exp a
  Add  :: Num a    ⇒ Exp a → Exp a → Exp a
class (Eq a, Num a) ⇒ Divide a where
  divide :: a → a → a
instance Divide Int where
  divide = div
instance Divide Double where
  divide = (/)
eval :: Exp a → Maybe a
eval (Int x)    = Just x
eval (Doub x)   = Just x
eval (Div a b) = do
  a′ ← eval a
  b′ ← eval b
  if b′ ≡ 0
    then Nothing
    else Just (a′ ‘divide‘ b′)
eval (Add a b) = do
  a′ ← eval a
  b′ ← eval b
  Just (a′ + b′)
```

(**Task**) By using type families, you should modify the EDSL so that the *Div* constructor can divide any combination of *Int*s and *Double*s. For instance, it is possible to compute *Div* (*Int* 10) (*Doub* 2.5) and *Div* (*Doub* 2) (*Doub* 2) in your language.

For the whole exercise, you can assume the function *fromIntegral* :: (*Integral a*, *Num b*) ⇒ *a* → *b*, which takes numbers with whole-number division and remainder operations (e.g., *Integer* and *Int*), and transformed them into numbers with basic operations (e.g., *Word*, *Integer*, *Int*, *Float*, and *Double*). *(7p)*

ii) The following code implements a type family (*Serialized*) and a type class (*Serialize*) which in combination are used for serializing data into tuples of words of a user-specified size. Observe that the type family works on two types.

```
type family Serialized t a where
  Serialized Word16 Int  = (Word16, Word16)
  Serialized Word16 Word = (Word16, Word16)
  Serialized Word8  Int  = (Word8, Word8, Word8, Word8)
  Serialized Word8  Word = (Word8, Word8, Word8, Word8)
    -- more cases (not relevant for the rest of the exercise)
class Serialize t a where
  serialize :: a → Serialized t a
instance Serialize Word16 Int where
  serialize i = (fromIntegral i, fromIntegral (i `shiftR` 16))
instance Serialize Word16 Word where
  serialize w = (fromIntegral w, fromIntegral (w `shiftR` 16))
  -- more instances (not relevant for the rest of the exercise)
```

Function *shiftR* shifts the first argument right by the specified number of bits.

The type family, type class and instances are all type-correct on their own. However, attempting to apply *serialize* to any value will cause a type error:

```
main = putStrLn ("High word: " ++ show hi)
  where
    lo, hi   :: Word16
    (lo, hi) = serialize (0xDEADBEEF :: Word)
```

This happens because *serialize* returns a type family application. In this case, the type of *serialize* is of the form *Word* → *Serialized t Word*. This makes the type checker unable to infer *t*, even though it is obvious that the *t* must be *Word16* in this case.

(**Task**) Explain *why* it is in general impossible to infer a type *t* even if we know what the type family application *F t* computes to. Think in the example above: why Haskell's type system does not choose *t* to be *Word16* when it sees that (*lo*, *hi*) has type (*Word16*, *Word16*)? The type error is as follows:

```
Couldn't match expected type (Word16, Word16)
                with actual type Serialized t0 Word
    The type variable t0 is ambiguous
    In the expression: serialize (3735928559 :: Word)
    In a pattern binding: (lo, hi) = serialize (3735928559 :: Word)
Failed, modules loaded: none.
```

(3735928559 is 0*xDEADBEEF* in the message above.) You should also describe *which additional properties* a type family definition would need to make the example above to type check, i.e. when Haskell sees *Serialized t Word*, it can infer that *t* must be *Word16*. *(7p)*

iii) To resolve problems like this, where the type checker does not have enough information to figure out what we want, it is common to use *proxy types*:

$$\textbf{data}\ \mathit{Proxy}\ a = \mathit{Proxy}$$

Proxies allow us to pass a type directly to a function, without having to come up with a concrete value of that type—we have the constructor *Proxy*! One instance where this is useful is when composing polymorphic functions, and we need to keep track of some intermediate result.

The following example will produce a type error, since there is no way for the compiler to infer the concrete return type of *read*, which makes impossible to choose a suitable parser from the dictionary *Read a*. More concretely, let us assume the following functions and definitions.

$$\begin{aligned}
&\mathit{read}\ :: \mathit{Read}\ a \Rightarrow \mathit{String} \to a\\
&\mathit{print} :: \mathit{Show}\ a \Rightarrow a \to \mathit{IO}\ ()\\
&\mathit{readAndPrint} :: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{readAndPrint} = \mathit{print} \circ \mathit{read}
\end{aligned}$$

We get the following type error:

```
No instance for (Read a0) arising from a use of read
    The type variable a0 is ambiguous
     In the second argument of (.), namely read
    In the expression: print . read
    In an equation for readAndPrint: readAndPrint = print . read
Failed, modules loaded: none.
```

By allowing the caller to explicitly provide a proxy with the return type of *read*, we can help the compiler to select the appropriated parser for *read*.

$$\begin{aligned}
&\mathit{read}' :: \mathit{Read}\ a \Rightarrow \mathit{Proxy}\ a \to \mathit{String} \to a\\
&\mathit{read}'\ p = \mathit{read}\\
&\mathit{readAndPrint}' :: (\mathit{Read}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Proxy}\ a \to \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{readAndPrint}'\ p = \mathit{print} \circ (\mathit{read}'\ p)
\end{aligned}$$

Observe that proxy $p :: \mathit{Proxy}\ a$ above is not used in the body of *read′*. It is there merely for having an argument which involves the returning type a. By instantiating a in *Proxy a*, we can indicate which parser must be used.

```
> readAndPrint' (Proxy :: Proxy Int) "42"
42
> readAndPrint' (Proxy :: Proxy Double) "1.42"
1.42
```

(**Task**) Use proxies to fix the *serialize* function from *ii)*. Then, write an example demonstrating how to use your fixed *serialize*. *(6p)*

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 1.

```
  -- 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 1: Allowed flows of information

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*.

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 2, and provide a *shallow-embedded* implementation for the API in Figure 3.

With our EDSL now, you can write functions which keep secrets! For instance, imagine a function which takes the salary of a employee in a certain position (sensitive information[1]) and determines if it is above the average.

$$isAbove :: Labeled\ H\ Salary \rightarrow Labeled\ L\ Salary \rightarrow MAC\ H\ Bool$$

Function *isAbove* takes the employee's salary (see argument of type *Labeled H Salary*) and the average (see argument of type *Labeled L Salary*) and returns a *MAC H*-computation indicating that the resulting boolean is sensitive—after all, it depends on the employee's salary! If the returning computation were *MAC L Bool*, then *isAbove* will not type-check: it would be impossible to unwrap the employee's salary using *unlabel*.

i) (**Task**) Take the EDSL and create a monad transformer for it, which we call *MACT*.

data *MACT l m 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*.

[1] In Sweden, salaries are public information but that is not the case in other countries.

```
  -- Types
newtype Labeled l a
newtype MAC l a
  -- Labeled values
label     :: (l `CanFlowTo` h) ⇒ a → MAC l (Labeled h a)
unlabel   :: (l `CanFlowTo` h) ⇒ Labeled l a → MAC h a
  -- MAC monad
return    :: a → MAC l a
(≫=)      :: MAC l a → (a → MAC l b) → MAC l b
joinMAC   :: (l `CanFlowTo` h) ⇒ MAC h a → MAC l (Labeled h a)
  -- Run function
runMAC    :: MAC l a → a
```

Figure 2: EDSL API

```
  -- Types
newtype Labeled l a = MkLabeled a
newtype MAC l a     = MkMAC a
  -- Labeled values
label                 = MkMAC ∘ MkLabeled
unlabel (MkLabeled v) = MkMAC v
  -- MAC operations
joinMAC (MkMAC t) = MkMAC (MkLabeled t)
runMAC (MkMAC a) = a
instance Monad (MAC l) where
  return = MkMAC
  MkMAC a ≫= f = f a
```

Figure 3: Shallow-embedded implemention

Define an implementation for $MACT\ l\ m\ a$ and give the type-signature and implementation of the following operations on transformed monads.

$$\begin{array}{ll}
\mathit{return} & :: \ldots \\
(\gg\!=) & :: \ldots \\
\mathit{t\_label} & :: \ldots \\
\mathit{t\_unlabel} & :: \ldots \\
\mathit{t\_joinMAC} & :: \ldots \\
\mathit{t\_runMAC} & :: \ldots
\end{array}$$

Help: We provide the type-signature of $\mathit{t\_label}$ and $\mathit{t\_runMAC}$.

$$\begin{array}{ll}
\mathit{t\_label} & :: (\mathit{Monad}\ m, l\ \text{`}\mathit{CanFlowTo}\text{`}\ h) \Rightarrow a \to \mathit{MACT}\ l\ m\ (\mathit{Labeled}\ h\ a) \\
\mathit{t\_runMAC} & :: \mathit{MACT}\ l\ m\ a \to m\ a
\end{array}$$

Observe that the type-signature looks almost similar to those in MAC where $MACT$ is used instead.

Hint: In the definition of $(\gg\!=)$, reuse as much as possible the monadic operators from monads m and MAC.

(10p)

ii) Assuming that m and MAC are monads, you need to prove that $MACT\ l\ m\ a$ is also a monad, i.e. you should show that your monad transformer generates monads! The monad laws are shown in Figure 4. In the proofs, you are likely to write the monadic operators *return* and $(\gg\!=)$. Since you would be dealing with more than one monad, it might get confusing to determine which monad you are referring to. Therefore, you must indicate as a subindex the name of the monad that operations refers to. For example, return_m, return_{MAC}, or return_{MACT} refers to the *return* operation for monad m, MAC, and $MACT$, respectively. Finally, if you need auxiliary properties, you should provide a proof for them, too!

a) Prove left identity. *(2p)*

b) Prove right identity. *(2p)*

c) Prove associativity. *(6p)*

Hint: You might need to prove an auxiliary property about $\mathit{t\_runMAC}$, $\gg\!=_m$, and $\gg\!=_{MACT}$.

Appendix

class *Monad m a* **where**
$\quad return :: a \to m\ a$
$\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b$

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

RIGHT IDENTITY
$m \gg\!= return \equiv m$

ASSOCIATIVITY (x DOES NOT APPEAR IN m_2 AND m_3)
$(m \gg\!= k_1) \gg\!= k_2 \equiv m \gg\!= (\lambda x \to k_1\ x \gg\!= k_2)$

Figure 4: Monads

FUNCTOR TYPE-CLASS
class *Functor c* **where** $fmap :: (a \to b) \to c\ a \to c\ b$

IDENTITY
$fmap\ id \equiv id$ **where** $id = \lambda x \to x$

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

Figure 5: Functors

APPLICATIVE TYPE-CLASS
class *Applicative c* **where** $pure :: a \to c\ a \qquad (\mathbin{<\!\!*\!\!>}) :: c\ (a \to b) \to c\ a \to c\ b$

IDENTITY
$pure\ id \mathbin{<\!\!*\!\!>} vv \equiv vv$ **where** $id = \lambda x \to x$

COMPOSITION
$pure\ (\circ) \mathbin{<\!\!*\!\!>} ff \mathbin{<\!\!*\!\!>} gg \mathbin{<\!\!*\!\!>} zz \equiv ff \mathbin{<\!\!*\!\!>} (gg \mathbin{<\!\!*\!\!>} zz)$

HOMOMORPHISM
$pure\ f \mathbin{<\!\!*\!\!>} pure\ v \equiv pure\ (f\ v)$

INTERCHANGE
$ff \mathbin{<\!\!*\!\!>} pure\ v \equiv pure\ (\$v) \mathbin{<\!\!*\!\!>} ff$

Figure 6: Applicative functors