Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Monday, August 26, 2019, "Maskin"-salar, 14:00-18:00.

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 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: (Optimization)

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

$$
\begin{aligned}
&average :: [Int] \to Int \\
&average\ xs = sum\ xs\ `div`\ length\ xs \\
&sum\ [\,] \quad\ = [\,] \\
&sum\ (x : xs) = x + sum\ xs \\
&length\ [\,] \quad\ = 0 \\
&length\ (x : xs) \ = 1 + length\ xs
\end{aligned}
$$

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 equation above. You should then implement *average* using *sumlen*.

(10p)

Problem 2: (**General questions**)

a) Is a monad a functor? If so, write *fmap* in terms of *return* and *bind*. Otherwise, give a counterexample of a monad that is not a functor.

(2p)

b) Is an applicative functor a functor? If so, write *fmap* in terms of *pure* and $(<*>)$. Otherwise, give a counterexample.

(2p)

c) Provide the code for a function of type $f :: a \to b$. *(2p)*

d) Assume two applicative functors $A\ a$ and $B\ a$ defined as follows.

```
newtype A a = MkA a
instance Functor A where
  fmap f (MkA a) = MkA (f a)
instance Applicative A where
  pure          = MkA
  (MkA f) <*> xx = fmap f xx
newtype B a = MkB a
instance Functor B where
  fmap f (MkB a) = MkB (f a)
instance Applicative B where
  pure          = MkB
  (MkB f) <*> xx = fmap f xx
```

We will now compose them in a single data type *Combined a* as follows.

```
newtype Combined a = MkC (B (A a))
```

Your task is to write an instance of *Applicative Combined* by using *pure* and $(<*>)$ from the applicative functors $A\ a$ and $B\ a$. *(4p)*

Problem 3: (Phantom types)

A phantom type is a parametrised type whose parameters do not *all* appear on the right-hand side of =. An example of such a type is the following.

newtype *Const a b* = *Const a*

Here *Const* is a phantom type since type parameter *b* does not occur in the implementation of the type. The idea of having "phantom" arguments (like *b*) above is commonly used to capture some invariant about the data contained by such a data type. Let us consider the following example. You are programming a web server and you know that forms accompanying web requests must be validated before processing them. To implement such an invariant, we introduce the following two empty types.

data *Unvalidated*
data *Validated*

Now, we declare the phantom type

data *FormData a* = *FormData String*

which uses *a* to indicate if the string has been validated. For instance, a string is initially considered as an unvalidated form.

formData :: *String* → *FormData Unvalidated*

Then, a validation function takes an *unvalidated* form into a possibly valid one.

validate :: *FormData Unvalidated* → *Maybe* (*FormData Validated*)

Once the string is validated, it can then be process.

useData :: *FormData Validated* → *IO* ()

a) You got a piece of code which manages different measurements with the following interface.

data *Measure* = *Measure Float*
measure_m :: *IO Measure*
measure_km :: *IO Measure*
add :: *Measure* → *Measure* → *Measure*
add (*Measure x1*) (*Measure x2*) = *Measure* (*x1* + *x2*)

However, you realize that measurements might be done in meters (*measure_m*), or kilometers (*measure_km*). Furthermore, the function *add* might add centimeters and kilometers, producing a measurement which has no sense. Your task is to modify the type signatures of the API above with phantom types to avoid mixing measurements of different units. Do you need to change the implementation of the function *add*?

(8p)

b) In the previous item, you realize that the function *add* can only add measurements of the same unit. To make the programming experience more smooth, you need to provide an overloaded version of *add* so that it can handle arguments of different units.

(4p)

c) As the phantom type *FormData a* was introduced in **a)**, it allows the type variable *a* to be instantiated to an arbitrary type. For instance, it could be instantiated to *Bool* and *Float*

fb :: *FormData Bool*
ff :: *FormData Float*

which has no meaning for the considered scenario. We would like to restrict *a* to be only instantiated to types *Unvalidated* and *Validated*. Your task is to write code such that the phantom type can only be instantiated to such types. Which extension of GHC do you need?

(8p)

Problem 4: (Faceted Values)

Sometimes software needs to handle information sensitive to different users. One way to ensure such separation is by using a *faceted* values semantics, where values have different facets (views) depending on who looks at them. The following data type implements faceted values.

```
type User = String
data Fac a where
  Raw  :: a → Fac a
  View :: User → Fac a → Fac a → Fac a
```

Term *Raw x* indicates that x is visible to all the users. Term *View u secret public* denotes the value *secret* if the program is allowed, or granted access, to look into u's data. In contrast, if it is explicitly forbidden from looking into u's data, then it will see *public*. If the program is neither allowed nor denied to look into u's data, then the view is undefined. Let us consider the following example:

```
example1 :: Fac Integer
example1 = View "Leonor" (Raw 42) (Raw 7)
```

If the program is allowed to see `"Leonor"`'s data, *example1* denotes 42. In contrast, if it is forbidden to do so, then *example1* denotes 7.

We call *projection* the function that takes a faceted value and extracts the value that it denotes depending on the permissions given to the program. More specifically,

```
type Allowed = [User]
type Denied = [User]
projection :: Fac a → Allowed → Denied → Maybe a
```

We assume that *projection* is always called with *disjoint* values of *Allowed* and *Denied*. This function returns a *Maybe*-value since it might not be possible to extract a value.

```
> projection example1 ["Leonor"] []
Just 42
> projection example1 []          ["Leonor"]
Just 7
> projection example1 ["Martin"] ["Leonor"]
Just 7
> projection example1 ["Martin"] []
Nothing
> projection example1 []          ["Martin"]
Nothing
```

The last two cases return nothing since it is neither allowed nor denied to see `"Leonor"`'s information.

Faceted values can also be nested, which implies that permissions might need to have more than one user in order to return a *Just*-constructor with a value. Let us consider the following example to illustrate that:

```
example2 :: Fac Integer
example2 = View "Leonor" (View "Martin"
                                (Raw 42)
                                (Raw 100)
                         )
                         (Raw 7)
> projection example2 ["Leonor", "Martin"] []
Just 42
> projection example2 ["Leonor"]          ["Martin"]
Just 100
> projection example2 ["Martin"]          ["Leonor"]
Just 7
> projection example2 ["Leonor"]          []
Nothing
> projection example2 []                  []
Nothing
```

a) Write the function *projection*. *(5p)*

b) In this part, we will build computations based on faceted values by giving it a monadic structure! For that, we start by extending our data type definition as follows to encode the monadic operations in a deep-embedded fashion.

```
data Fac a where
  Raw  :: a → Fac a
  View :: User → Fac a → Fac a → Fac a
  Bind :: Fac a → (a → Fac b) → Fac b
```

If *Fac* is a monad, then programs can safely manipulate faceted values for different users' views without compromising their confidentiality. For instance, the following code

```
p1 = example1 >>= λx → return (x + 1)
p2 = example2 >>= λx → return (x + 1)
```

denotes the faceted value *View* `"Leonor"` (*Raw* 43) (*Raw* 8) — observe that the bind applies $x + 1$ to every leaf in the tree-structure denoted by faceted values. More concretely,

```
> projection p1 ["Leonor"] []
Just 43
> projection p1 []          ["Leonor"]
Just 8
> projection p2 ["Leonor"] ["Martin"]
Just 101
```

Give an instance of the type-class *Monad* for *Fac* and extend your definition of *projection* to consider *Bind*. *(7p)*

c) Transform your code to implement *Bind* in a shallow-embedded fashion. *(8p)*