Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Monday, August 21, 2017, "Maskin"-salar, 14:00 (4hs)

Alejandro Russo, tel. 0729-744-968

- 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)*

Problem 1: (**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* x_1) (*Measure* x_2) = *Measure* ($x_1 + x_2$)

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)

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 1: Functors

Problem 2: (**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**
  $Leaf :: a \to Tree\ a$
  $Node :: Tree\ a \to Tree\ a \to Tree\ a$

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).

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., on the length of the list). Justify every step in your proof.

(8p)

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)*

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 extract the leaves (right-hand side).

Your task is to prove that the equation holds.

You can assume the following properties and definitions 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)*

Problem 3: (**Miscellaneous**)

a) Write a function of type $a \to b$, where a and b are polymorphic types. *(4p)*

b) Let us consider the alternative formulation of monads.

$$\begin{aligned} &\mathit{return}' :: a \to m\ a \\ &\mathit{join} \quad\ :: m\ (m\ a) \to m\ a \\ &\mathit{fmap} \quad :: (a \to b) \to m\ a \to m\ b \end{aligned}$$

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 involving *return* and $(\gg\!=)$.

$$\begin{aligned} m \gg\!= f \quad &\equiv \mathit{join}\ (\mathit{fmap}\ f\ m) \\ m \gg\!= \mathit{id} \quad &\equiv \mathit{join}\ m \\ m \gg\!= (\mathit{return} \circ f) \quad &\equiv \mathit{fmap}\ f\ m \end{aligned}$$

Now, let us consider the function $(+)$ with the following type signature:

$$(+) :: \mathit{Num}\ a \Rightarrow a \to a \to a$$

What is the output of *join* $(+)$? Hint: remember the reader monad.

(8p)

c) Give an example of a data type definition which is not a functor and explain why this is the case. You should be clear why *fmap* cannot be implemented.

(4p)

d) Define a data type that is a functor and not applicative. Define *fmap* and justify why the instance *Applicative* cannot be defined.

(4p)