Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Monday, March 18, 2023, 08:30.

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 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 (20pt): (The free monad)

The free monad is a structure that enables getting a monadic instance from a functor. So, if non-proper morphism can be expressed as functors, then the definition of *return* and ($\gg\!=$) are for free! Free here means that we do not need to re-define *return* and ($\gg\!=$) for every new non-proper morphism that we wish to add. They are defined once and for all. Free monads bring modularity for defining DSL interpreters, and we will see some of that in the last exercise.

For a given functor f, the free monad is defined as follows:

```
data Free f a = Pure a | NonProper (f (Free f a))
```

Instructions in the free monad are either a value a (*Pure* a) or a non-proper morphism – encoded by functor f—which considers a monadic computation (*Free* f a).

To illustrate how free monads work, we consider a DSL to read and write strings from the terminal. You can see the interface of the DSL below – it is pretty similar to what we have seen during the course.

```
  -- Type for programs performing I/O actions
data Console a
  -- Constructors
return :: a → Console a
print :: String → Console ()
input :: Console String
  -- Combinators
(>>=) :: Console a → (a → Console b) → Console b
  -- Run function
evalC :: Console a → IO a
```

So, we start by defining the non-proper morphisms as a functor:

```
data ConsoleF m
  = Print String m
  | Input (String → m)
  deriving (Functor, Applicative)
```

Intuitively, the way to model non-proper morphism as a functor is by following roughly the next two rules:

a) Add one constructor for each non-proper morphism.

b) Abstract away the type of the monadic computation. In the example above, when you see something of type *Console* a, replace it with a computation m where m is a type variable.

c) If a non-proper morphism takes an argument, give that argument to the constructor representing it. For instance, *print* :: *String* $\to$ *Console* () has the argument of type *String*, so we have the constructor *Print String* m. The way to think about it is that *Print* represents a computation that, *given a string, it will produce an output and then continue as the computation* m.

d) If a non-proper morphism returns a monadic value, then the constructor should receive a function that waits for that value and then indicates how to continue the computation. For instance, *input* :: *Console String* produces a monadic value of type *String*, so the constructor *Input* takes a function of type *String* $\rightarrow m$ as an argument.

We then define a free monad instance using *ConsoleF*:

```
type Console a = Free ConsoleF a
```

This definition will use *return* and ($\gg=$) of the free monad, thus being able to write code like:

```
program :: Console ()
program = do
  print "What is your name?"
  n ← input
  print $ "Hello! " ++ n
```

When we run the program, we get

```
*> evalC program
What is your name?
 Leonor
Hello! Leonor
```

Do now worry about the interpreter and the accompanying functions, we will get there later in the exam. For now, we will start by defining more elementary functions for the free monad.

a) Your task is to define ‘fmap‘ for the free monad.

```
instance Functor f ⇒ Functor (Free f) where
```

(5p)

b) Your task is to define the applicative instance for the free monad.

```
instance Applicative f ⇒ Applicative (Free f) where
```

(7p)

c) Your task is to give a monadic instance for the free monad – observe that this instance will be generic on the functor f being considered. So, the *return* and ($\gg=$) definitions that you will provide will be useful when considering any functor f, and thus the generality.

```
instance Applicative f ⇒ Monad (Free f) where
```

(8p)

Problem 2 (25pt): (Interpreters with the free monad)
We will get back now to the example of the DSL.

a) To write any useful program, we need a notion of lifting of the following type:

$$\mathit{liftF} :: \mathit{Functor}\ f \Rightarrow f\ a \rightarrow \mathit{Free}\ f\ a$$

which takes a value constructed with the function f and injects it into the free monad. Your task is to provide the definition of *liftF* *(8p)*

b) Using the *liftF* function and *ConsoleF*, write the definition for

$$\mathit{print} :: \mathit{String} \rightarrow \mathit{Console}\ ()$$

(5p)

c) Using the *liftF* function and *ConsoleF*, write the definition for

$$\mathit{input} :: \mathit{Console}\ \mathit{String}$$

(5p)

d) You need to write now a run function for the *Console* monad.

$$\mathit{evalC} :: \mathit{Console}\ a \rightarrow \mathit{IO}\ a$$

The idea is that when hitting a non-proper morphism *getLineFM* and *putStrLnFM*, the run function invokes *getLine* and *putStrLn* from the *IO*-monad, respectively.

In this exercise, you need to complete the ... in the given code skeleton:

```
evalC (Pure x)          = ...
evalC (NonProper fx) = case fx of
                          Print o m → ...
                          Input f   → ...
```

(7p)

Problem 3 (15pt): (Extending interpreters)

a) We will now remove the pattern-matching from the run function in Exercise 2d. For that, you are given a function that works on the functor *ConsoleF*:

```
sem :: ConsoleF (IO a) → IO a
sem (Print o r) = Prelude.putStrLn o ≫ r
sem (Input f)   = Prelude.getLine ≫= f
```

Using *sem*, rewrite *evalC* and remove the pattern-matching. By doing that, all the semantics of the non-proper morphisms is captured in the function *sem*. *(5p)*

b) Extend *ConsoleF* and the rest of your code to consider a new non-proper morphism *random* :: *Console Int* responsible to fetch a random number by using *randomIO* :: *IO Int*.

You should be able to run the following program:

```
program :: Console ()
program = do
  print "What is your name?"
  n ← input
  print $ "Hello! " ++ n
  print $ "And here is a random integer:"
  i ← random
  print $ show i
```

that produces

```
*> evalC program
What is your name?
  Leonor
Hello! Leonor
And here is a random integer:
3588367089568947562
```

Did you need to change *return* and (≫=)? Were your changes rather localized to some specific parts of the code?

(10p)