Online Exam: Advanced Functional Programming TDA342/DIT260

The whole exam is in a `.pdf` file in case you have connection problems. Search for the file `exam.pdf` and **download it as the first thing to do!**

As the second thing, **download the file `exam-0.1.0.0.tar.gz`**

Date	Saturday, 20 March 2020
Time	8:30 - 12:30 (4hs)
Submission window	12:30 - 12:40

Preliminaries

Below you find the information that is often on the **first page of a paper exam**

- Remember to take ten minutes to submit your online exam. There is a submission window (12:30 - 12:40), but if you finish the exam before, you can submit it as you did when submitting regular assignments ahead of time.
- The maximum amount of points you can score on the exam: 60 points. The grade for the exam is as follows:
  - Chalmers students:
    - 3: 24 - 35 points
    - 4: 36 - 47 points
    - 5: 48 - 60 points
  - GU students:
    - Godkänd: 24-47 points
    - Väl godkänd: 48-60 points
  - PhD student: 36 points to pass.

- Results will be available within 21 days from the exam date.
- Notes:
  - Read through the exam 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.
  - *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*)

Preliminaries about this online exam

Below you find the information that is related to **the online exam**

- Please note that this is an exam to be carried out individually, and since this is an exam from home, we will be very strict with plagiarism.
- This exam considers that you will have open books as well as Internet access, i.e., access to the course's content and code, **Hoogle** , etc. -- and you can use all of such resources!
- The exam consists on *programming exercises*.
  - You will get a source code skeleton for each coding exercise that you need to complete.
- The exam is designed to **not need any special Haskell package**. It is enough to use the ones imported by each source file.

What and how to submit

- You should submit the code skeleton **completed with your solution**. The name of the code skeleton can be found below.

Code skeleton	**exam-0.1.0.0.tar.gz**

- Before you submit your code, please clean it up! We will use the same requirements for clean code as in the course's assignments, that is, clean code
  - does not have long (> 80 characters) lines
  - has a consistent layout
  - has type signatures for all top-level functions
  - has good comments for all modules, functions, data types, and instances.
  - has no junk (junk is unused code, code which is commented out, unnecessary comments)
  - has no overly complicated function definitions
  - does not contain any repetitive code (copy-and-paste programming)
- **Use `cabal sdist` to generate the source tarball that you will submit**. (**You can also use `stack` to build it if you prefer so**). Make sure the tarball is working by extracting it in another directory and running `cabal configure` and `cabal build` and check that everything looks right.
- **In addition, submit the files `Ex1.hs`, and `Ex2.hs` with your solutions in Canvas and make sure that they are not part of any directory/folder**, i.e., we want just plain `.hs` files. This helps us to grade your submission fast since Canvas does not understand `.tar.gz` files.

Questions during the exam

We will use Zoom for doing the "exam rounds" to answer questions. There will be two rounds, one at 9:30 and another at 11:30. Please, install and be familiar with Zoom (https://chalmers.zoom.us/ . Below, the instruction to follow:

1. You will join the meeting in a waiting room and need to wait until it is your turn.

2. I will grab one by one into the meeting to answer your questions.

Below, the invitation for the Zoom meeting:

Round 1 (9:30)

```
https://chalmers.zoom.us/j/65241135824 Time: Mar 21, 2020 9:30 AM Stockholm
```

Round 2 (11:30)

```
https://chalmers.zoom.us/j/67842740891  Time: Mar 21, 2020 11:30 AM Stockholm
```

Contingency plan (if things fail)

Exam rounds

- If Zoom fails: send your question as a message in Canvas to Alejandro Russo
- If Canvas fails: send your questions by email to Alejandro Russo

Submission of the exam

- If Canvas fails: send your exam by email to Alejandro Russo, please use the following format in the subject `[TDA342/DIT260 Exam, your personal number, your name]`

Good luck!

Exercise 1 (25 points)

File `src/Exam/Ex1.hs`

In this exercise, you will learn about a special kind of (popular) monads: *graded monads*! Graded monads are monads indexed by a monoid. A monoid consists of a binary operation $\oplus : E \rightarrow E \rightarrow E$ that is associative and has an "empty"l element $\epsilon \in E$. The monoid structure is good to represent programs with effects, but we will get there later! More formally, a monoid fulfill the following properties:

- $(o \oplus p) \oplus q = o \oplus (p \oplus q)$
- $a \oplus \epsilon = \epsilon \oplus a = a$

The idea is that graded monadic values have type $m_o a$ where $o \in E$. In other words, you have a family of monads m, where you have one member in the family for every element in the domain of the monoid.

The return and the bind operators for graded monads are defined as follows:

- $\text{return} : a \to m_\epsilon a$
- $(\gg\!=) : m_o a \to (a \to m_p b) \to m_{o \oplus p} b$

Observe that the bind applies the monoidal operator to the index of the first and second monadic operation.

Task 1.1 (5 pts)

In order to start implementing graded monads, we first need to provide a data type that represents a monoid on elements of a certain type:

```
> data Monoid t = Empty | Elem t | Plus (Monoid t) (Monoid t)
```

Constructor `Empty` represents the empty element of the monoid. The constructor `Elem t` denotes an element of the monoid (of type `t`). Constructor `Plus` represent the $\oplus$ operators. *We will assume that the monoid properties hold in the code -- the code does not need to check them!*

Your task is to encode graded monads in Haskell. More specifically, you need to provide a graded monad transformer GMonad, which index a monad m with a provided type level element o from the monoid. That is, $\mathrm{GMonad}\ o\ m\ a$ should denote $m_o a$.

```
data GMonad o m a = ?
```

Please, check the type declaration and make sure that `o` represent an element of the monoid, i.e., **it is a type of kind** `Monoid t` **for some** `t`. Here, you need to use `DataKinds` and type-level programming aspects of Haskell (you can find them explained in the type-based modeling module of the course or in this link).

Task 1.2 (5 pts)

In this task you need to define the return and bind operators for graded monads. You cannot use Haskell's `Monad` class, the reason being that it does not support indexed monads where the index gets change by the bind! Recall that the bind for monads is

$(\gg=) : m\ a \to (a \to m\ b) \to m\ b$, while the bind for graded monads is $(\gg=) : m_o a \to (a \to m_p b) \to m_{o \oplus p} b$.

In this task you need to give the type signature and implement:

```
returnGMonad -- the return for graded monads
bindGMonad   -- the bind for graded monads
```

You also need to implement a "run" function that takes you back from the graded monad to the underlying one.

```
runGMonad :: GMonad o m a -> m a
```

Task 1.3 (15 pts)

In this task, you need to prove that graded monads are monads, that is, they fulfill the monadic laws.

- $\mathit{return}\ x \gg= f = f\ x$
- $m \gg= \text{return} = m$
- $(m \gg= f) \gg= g = m \gg= (\lambda x.\, f\ x \gg= g)$

You should do the proof as comments inside the file. For instance, look for the place where you should write the proof of the first monadic law:

```
Law 1:
return x >>= f =?= f x

   return x >>= f
== (justification)
   ...
== (justification)
   f x
```

As you can see, you need to fill the dots and justify each step. For instance, we can complete the first step of this proof as follows:

```
Law 1:
return x >>= f =?= f x

   return x >>= f
== (def. return and bind for graded monads)
   bindGMonad (returnGMonad x) f
== (justification)
   ...
== (justification)
   f x
```

You should be precise at each step in the proofs and it might happen that you need to prove an auxiliary lemma.

Note: If you have chosen to use a deep embedding to represent graded monad, the monadic laws above need to be slightly rephrased to involve `runGMonad`.

Exercise 2 (35 points)

File `src/Exam/Ex2.hs`

In this exercise, we will use a graded monad to control how resources are used by a monadic computation. *We will start focusing on imposing a limit on the number of writes to files on IO computations -- you could think that such a policy is useful for cloud environments or embedded devices with constraints power.* From now on, all the ideas and intuition are going to be given following mathematic notation **and not code -- you will need to write the code**.

Since we need to control the number of writes, we will use type-level natural numbers of kind `Nat` as provided by GHC -- see `import GHC.TypeLits` in `src/Exam/Ex2.hs`, which provides a number of type level natural numbers and operations (*type families*), e.g., `1 :: Nat, 2 :: Nat, 1 + 2 :: Nat, 3 <= 4`, etc.

Task 2.1 (5 pts)

We are going to choose `IO` as the underlying monad, and the natural numbers as the monoid elements, addition as the monoid operation, and `0 :: Nat` as the empty element. So, we provide write to files operations which register in the index of the monad that they perform a write:

- $\text{sWriteFile} :: \text{FilePath} \to \text{String} \to \text{IO}_1 ()$

This function is a non-proper morphism, that is, it is a primitive operation! Now, we can write the following code:

- $\text{twoWrites} = \text{sWriteFile}\ \texttt{"file1"}\ \texttt{"hello"} \gg \text{sWriteFile}\ \texttt{"file2"}\ \texttt{"bye"} :: \text{IO}_{1+1} ()$

Observe how the index of the graded monad keeps tracks of the amount of writes done by the computations!

Your task is to write the function sWriteFile using `GMonad` and twoWrites by using `bindGMonad`. It should be the case that when you run twoWrites with `runGMonad`, it indeed writes into two files.

Task 2.2 (5 pts)

It is the case that when you use `bindGMonad`, you will end up with a *type-level expression* reflecting that each step of the bind applies the monoid operator. For instance,

- $\text{fourWrites} = \text{twoWrites} \gg \text{twoWrites} :: \text{IO}_{(1+1)+(1+1)} ()$

Write a *type family* called `Norm` such maps *a type of kind monoid* (like $(1+1)+(1+1)$) to *a type of kind* `Nat` (like the type 4).

Hint: Do not write a type family `Norm` that works for any monoid, just focus on a monoid where the elements are natural number types.

Task 2.3 (5 pts)

We want now to introduce a predicate that makes assertions about the number of writes a computation can do.

- $\text{maxWrites}\ n\ (m_o\ a) = o \leq n$

The predicate works on a natural number (n) and a graded monad type ($m_o a$). Your task is to provide the primitives (and type-signature) `assertMaxWrites` that implement the above predicate at the type-level. In that manner, you can write code like the following:

```
okWrites = assertMaxWrites (Proxy :: Proxy 2) twoWrites
```

Code `okWrites` does the same computations as `twoWrites`, *and type-checks! So, you will have the compile-time guarantees that there are never more than two calls to* `sWriteFile` *in* `twoWrites.` *However, the following code should not type-check:*

```
badWrites = assertMaxWrites (Proxy :: Proxy 1) twoWrites
```

Hint: You will need

- *the type family defined in the previous exercise,*
- ***Proxy types*** ,
- ***GHC.TypeLits*** *and* ***GHC.TypeNats*** ,
  You might need the unification type constraint, e.g., `1 ~ a` *to indicate the Haskell type-system that type variable* `a` *should be unified (equal) to the type-level natural number* `1`

Task 2.4 (5 pts)

Writing into a file is not the only effect we could control. For instance, we could also control the reading effects! Previously, the monoid structure counted the number of writes. In this task, we consider a monoid structure that counts both, i.e., reads and writes. For that, we need a monoid that works on pairs of natural numbers, where the first component is the number of reads and the second component is the number of writes. More formally,

- $\oplus : (Nat, Nat) \to (Nat, Nat) \to (Nat, Nat)$
- $\epsilon = (0, 0)$

So, we consider the following *non-proper morphisms* to read and write files:

- $\text{sReadFile}' :: \text{FilePath} \to \text{IO}_{(1,0)}\text{String}$
- $\text{sWriteFile}' :: \text{FilePath} \to \text{String} \to \text{IO}_{(0,1)}()$

Observe how changing the monoid, we can know track more effects! *As we did before, we will assume that the monoid properties hold in the code -- the code does not need to check them!*

Your task is to provide an implementation of $\text{sReadFile}'$ and $\text{sWriteFile}'$ using `GMonad`. For that, you only need to change where the elements from the monoid come from when using `GMonad`. *You should be able to implement the following code:*

$\text{twoWritesOneRead} :: \text{IO}_{(0,1)+((1,0)+(0,1))}\,()$

$\text{twoWritesOneRead} = \text{sWriteFile}'\ \texttt{"file1"}\ \texttt{"hello"} \gg \text{sReadFile}'\ \texttt{"file2"} \ggg\!= (\lambda s \to \text{sWriteFile}'\ \texttt{"file3"}\ s)$

Task 2.5 (5 pts)

In this task, you need to implement the type family `Norm2` which maps *a type of kind monoid on pairs of naturals* (like $(1,0)+(0,1)$) to *a type of kind* `(Nat,Nat)` (like type $(1,1)$).

Recall that you should not use the type constructor `(,) :: * -> * -> *`. *You cannot create the type-level pair* `(1,2)` *of kind* `(Nat,Nat)` *since type-level number* `1` *has kind* `Nat` *and not* `*` *as required by type constructor* `(,) :: * -> * -> *`*. However, you can use* `'(,) :: k1 -> k2 -> (k1,k2)` *(where* `k1` *and* `k2` *are kinds) to create a kind representing pairs, e.g.,* `'(1,2)` *is a type-level pair of kind* `(Nat,Nat)`.

Task 2.6 (5 pts)

We want to introduce predicates that make assertions about the maximum number of reads and writes a computation can do.

- $\text{maxWrites}'\ n\ (m_{(r,w)}\ a) = w \leq n$
- $\text{maxReads}'\ n\ (m_{(r,w)}\ a) = r \leq n$

The predicate takes a natural number and a graded monadic computation where the monoid is a pair of natural numbers.

Your task is to provide the primitives (and type-signature) of `assertMaxReads'` *and* `assertMaxWrites'` that implement the above predicate the type-level, respectively. In that manner, you can write code like the following:

```
good =  assertMaxWrites' (Proxy :: Proxy 2)
      $ assertMaxReads'  (Proxy :: Proxy 2)
      $ twoWritesOneRead
```

Observe that code `good` does the same computations as `twoWritesOneRead`*, and type-checks! So, you will have the compile-time guarantees that there are never more two calls to* `sReadFile'` *and no more two calls to* `sWriteFile'``.` *However, the following code should not type-check:*

```
notgood1 = assertMaxWrites' (Proxy :: Proxy 2)
         $ assertMaxReads'  (Proxy :: Proxy 0)
         $ twoWritesOneRead
notgood2 = assertMaxWrites' (Proxy :: Proxy 1)
         $ assertMaxReads'  (Proxy :: Proxy 2)
         $ twoWritesOneRead
```

Hint: you need t*he unification type constraint, e.g.,* `1 ~ a` *to indicate the Haskell type-system that type variable* `a` *should be unified (equal) to the type-level natural number* `1`.

Task 2.7 (5 pts)

So far, we were not able to use the do-notation of GHC. The reason is that, as we saw in the lecture, the do-notation desugars to the `return` and `(>>=)` *of the type class* `Monad`*. Would it be nice if GHC could desugar the do-notation instead to* `returnGMonad` *and* `bindGMonad`*? GHC can do that thanks to the extension* ***Rebindable Syntax*** *!*

Use this extension to enable do-notation in your implementation. In that manner, your code can now look like this:

```
twoWritesOneRead = do
  sWriteFile' "file1" "hello"
  s <- sReadFile' "file2"
  sWriteFile' "file3" s
```

Hint: remember to import Predule but hiding the standard `return` *and* `(>>=)`*, i.e.,* `import Prelude hiding ((>>), (>>=), return, Monoid)`

↓↓↓ SUBMISSION FORM ↓↓↓

This tool needs to be loaded in a new browser window

Load Online Exam: Advanced Functional Programming TDA342/DIT260 in a new window